

IBM z/OS Container Extensions (zCX) Use Cases

Lydia Parziale

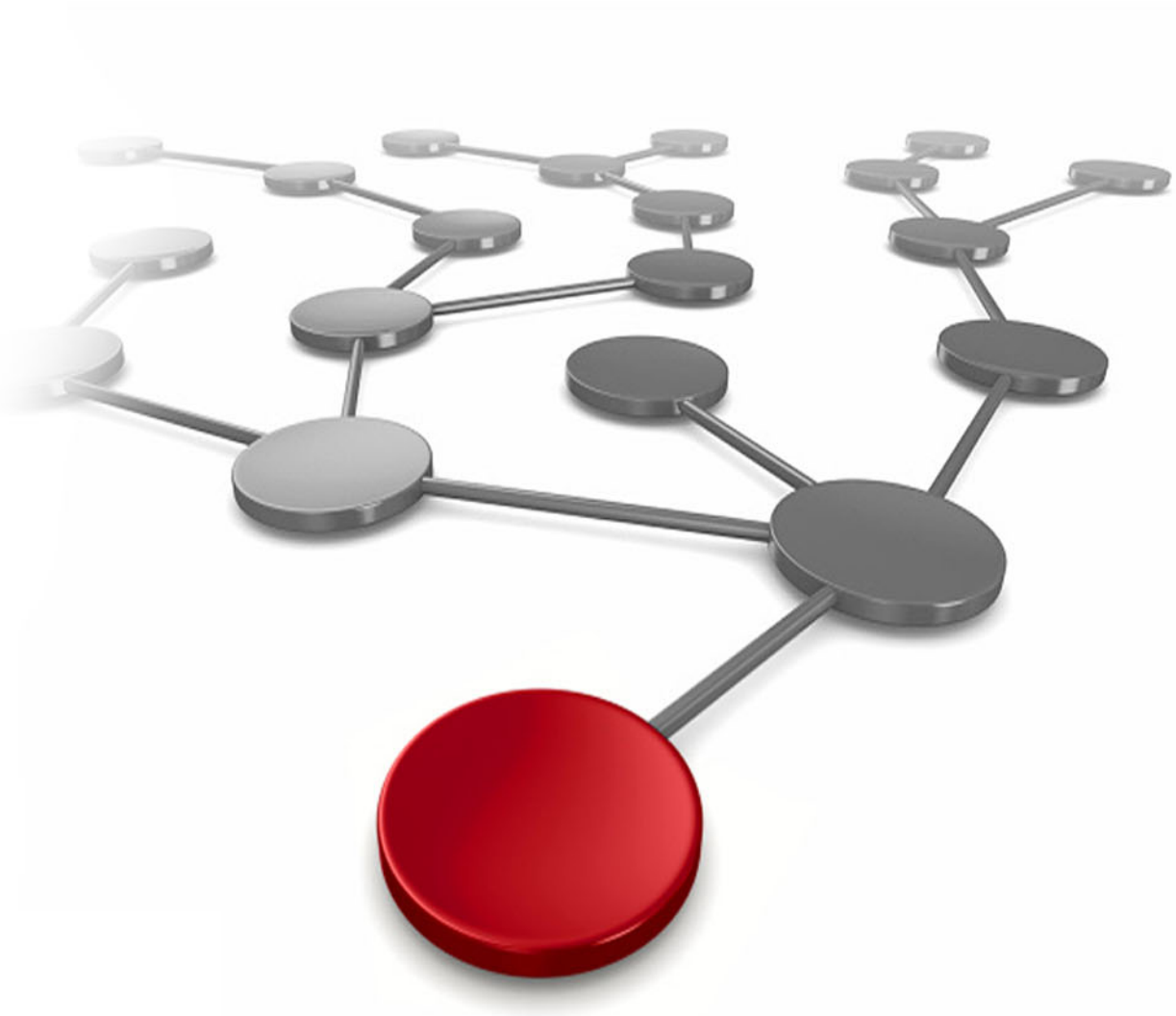
Marco Egli

Maike Havemann

Subhajit Maitra

Eric Marins

Edward McCarthy





IBM Redbooks

IBM z/OS Container Extensions (zCX) Use Cases

October 2020

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

First Edition (October 2020)

This edition applies to Version 2, Release 4, z/OS, Product number 5650-ZOS.

© Copyright International Business Machines Corporation 2020. All rights reserved.

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

Contents

Notices	vii
Trademarks	viii
Preface	ix
Authors	ix
Now you can become a published author, too!	x
Comments welcome	xi
Stay connected to IBM Redbooks	xi
Chapter 1. IBM z/OS Container Extensions overview	1
1.1 Why use IBM z/OS Container Extensions?	2
1.2 IBM zCX architecture	2
1.3 zCX updates	3
1.3.1 zCX 90-day trial	4
1.3.2 Docker proxy certificate support	4
1.3.3 IBM License Manager Tool support	5
1.3.4 Performance Improvements (zCX and z/OS Communication Server)	6
1.3.5 Private CA certificate support for Docker proxy	7
1.3.6 Vector / SIMD Support	7
Part 1. Integration	9
Chapter 2. Apache Kafka and ZooKeeper	11
2.1 ZooKeeper overview	13
2.2 ZooKeeper configuration	13
2.2.1 Persistence with Docker volumes	13
2.2.2 ZooKeeper default service ports	14
2.2.3 ZooKeeper configuration file	14
2.3 Kafka overview	15
2.4 Kafka configuration	17
2.4.1 Persistence with Docker volumes	17
2.4.2 Kafka configuration file	17
2.5 Kafka configuration on one zCX instance	18
2.5.1 Creating the Kafka cluster	18
2.6 Kafka configuration on three zCX instances	39
2.7 Diagnostic commands	60
2.7.1 ZooKeeper	60
2.8 Integrating IBM Z applications and Kafka	61
2.8.1 Event-driven application and Apache Kafka	61
2.8.2 Key usage patterns of Apache Kafka and IBM Z	62
2.8.3 Unlocking data from IBM Z to Kafka	63
2.9 Integration of IBM MQ with Apache Kafka	64
2.9.1 Setting up IBM MQ connectors to run on IBM z/OS	64
2.9.2 Starting Kafka Connect on z/OS	70
2.9.3 Status of plug-ins and connectors	70
2.9.4 Sending messages from IBM MQ to Kafka	71
2.10 Sending CICS events to Kafka	74
2.10.1 Why CICS events?	74
2.10.2 CICS to Kafka overview	74

2.10.3	Example components	76
2.10.4	ZKAFKA Java program	77
2.10.5	Testing the example	78
2.10.6	z/OS Explorer	80
2.10.7	Developing the ZKAFKA Java program	84
2.10.8	Creating CICS bundle for Java program	91
2.10.9	Defining a CICS event	98
Chapter 3.	IBM App Connect Enterprise	103
3.1	Technical and architectural concepts of ACE	104
3.1.1	Key concepts of ACE	106
3.1.2	Runtime components of ACE	107
3.1.3	ACE run time in zCX	108
3.1.4	Reasons to run ACE on zCX	109
3.2	Installing IBM App Connect Enterprise	109
3.2.1	Obtaining the ACE installation binaries	109
3.2.2	Obtaining the IBM App Connect Enterprise Docker container build source	111
3.2.3	Building the Dockerfile for the ACE image	114
3.2.4	Building the ACE Docker image	116
3.3	Configuration details	118
3.4	Deploying an application to ACE to integrate with CICS	119
3.4.1	Deploying to ACE run time in zCX	122
3.4.2	Using the Web UI to test deployed REST APIs	125
Chapter 4.	IBM Aspera fasp.io Gateway	129
4.1	Introduction to Aspera FASP.io Gateway	130
4.2	Aspera configuration details	131
4.2.1	Configuration Part 1: fasp.io on the first zCX (sc74cn09) instance	132
4.2.2	Configuration Part 2: fasp.io on the second zCX (sc74cn04) instance	137
4.3	Integration with IBM MQ Advanced for z/OS, VUE	141
4.3.1	Topology	141
4.3.2	Creating a transmission queue on MQZ1	141
4.3.3	Creating sender channel on MQZ1	142
4.3.4	Creating receiver channel on MQZ2	142
4.3.5	Creating transmission queue on MQZ2	142
4.3.6	Creating sender channel on MQZ2	142
4.3.7	Creating receiver channel on MQZ1	143
4.3.8	Verifying the infrastructure and configuration	143
4.3.9	Verifying two-way communication	144
4.3.10	Benefits of running the FASP.IO gateway on zCX	146
Chapter 5.	Using IBM MQ on zCX as a client concentrator	147
5.1	Interconnecting IBM MQ on z/OS with IBM MQ in zCX	148
5.1.1	Architecture	149
5.1.2	Scenario	149
5.1.3	Downloading an IBM MQ image inside zCX from dockerhub	150
5.1.4	Creating the queue manager inside the container	152
5.1.5	Creating IBM MQ channels between the queue manager on zCX and z/OS	154
5.1.6	Testing the message flow from both directions	157
5.2	Using IBM MQ on zCX as a client concentrator	160
5.2.1	IBM MQ clients	160
5.2.2	IBM MQ client concentrator	162
5.2.3	IBM MQ on zCX as a client concentrator	164

Part 2. DevOps in zCX	165
Chapter 6. DevOps overview	167
6.1 What is DevOps?	168
6.2 Why containerize DevOps?	169
6.3 Why use DevOps in zCX?	169
Chapter 7. Using Gitea as a code repository	171
7.1 Setting up Gitea in zCX	172
7.1.1 Building a Gitea Docker image	172
7.1.2 Setting up a private registry	174
7.2 Running a Gitea Docker container	175
7.3 Checking the Gitea Docker container status	176
7.4 Configuring Gitea	177
7.4.1 Uploading code to Gitea	179
Chapter 8. Using Jenkins to automate builds	183
8.1 What is Jenkins?	184
8.2 Building a Jenkins server Docker image	184
8.3 Running a Jenkins server Docker container	187
8.4 Checking Jenkins server Docker container status	188
8.5 Configuring Jenkins	188
8.5.1 Installing plug-ins manually	194
8.6 Setting up a Jenkins build agent	196
8.6.1 Creating a Jenkins build node	197
8.6.2 Building a Jenkins agent Docker image	200
8.7 Running a Jenkins build agent Docker container	203
8.8 Checking Jenkins build agent Docker container status	204
Chapter 9. Using Ansible to automate deployment and tests	205
9.1 Ansible overview	206
9.2 Setting up Ansible by using Jenkins	206
9.2.1 Creating an Ansible playbook for deployment to a development environment	206
9.2.2 Creating an Ansible playbook for integration tests	207
9.2.3 Creating an Ansible playbook for deployment to a test environment	207
9.2.4 Creating a deployment agent for Ansible in Jenkins	208
9.3 Setting up a Jenkins deployment agent	208
9.3.1 Creating a Jenkins deployment node	208
9.4 Running a Jenkins deploy agent Docker container	210
9.5 Checking Jenkins deploy agent Docker container status	211
Chapter 10. Putting it all together and running the pipeline	213
10.1 Pipeline architecture	214
10.2 Setting up a pipeline in Jenkins	215
10.3 Creating a webhook in Gitea	220
10.4 Running a pipeline in Jenkins	223
10.5 Monitoring a pipeline in Jenkins	225
10.6 Summary	227
Part 3. Monitoring and managing zCX systems	229
Chapter 11. Monitoring	231
11.1 IBM Service Management Unite	232
11.1.1 IBM Service Management Unite Automation	232
11.2 IBM Service Management Unite overview	233

11.3	Installation steps	235
11.4	Configuring the network and port	236
11.5	Starting the SMU image	237
11.6	Uninstalling Service Management Unite	241
11.7	Command wrapper overview	243
11.7.1	SMU Docker command line utility commands.	244
Chapter 12.	Container restart policies	247
12.1	Docker restart policies introduction	248
12.2	Summary.	251
Appendix A.	Additional material	253
	Locating the web material	253
	Using the web material.	253
	Downloading and extracting the web material	254
Related publications	255
	IBM Redbooks	255
	Online resources	255
	Help from IBM	255

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.

AIX®	IBM®	NetView®
Aspera®	IBM API Connect®	Redbooks®
CICS®	IBM Cloud®	Redbooks (logo)  ®
CICS Explorer®	IBM Z®	System z®
DB2®	IBM z15™	WebSphere®
Db2®	InfoSphere®	z/OS®
FASP®	Jazz®	z15™

The following terms are trademarks of other companies:

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

The registered trademark Linux® is used pursuant to a sublicense from the Linux Foundation, the exclusive licensee of Linus Torvalds, owner of the mark on a worldwide basis.

Microsoft, 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.

Ansible, OpenShift, Red Hat, are trademarks or registered trademarks of Red Hat, Inc. or its subsidiaries in the United States and other countries.

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

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

Preface

Is it time for you to modernize your IBM® z/OS® applications to allow for access to an entire system of open source and Linux on IBM Z® workloads? Is co-location of these workloads on the z/OS platform with no porting requirements of value to you?

Your open source or Linux on IBM Z software can benefit from being co-located and managed inside a z/OS environment; leveraging z/OS quality of service for optimized business continuity.

Your software can be integrated with and can help complement existing z/OS workloads and environments. If your software can communicate with z/OS and external components by using TCP/IP, now is the time to examine how IBM z/OS Container Extensions (IBM zCX) makes it possible to integrate Linux on Z applications with z/OS.

This IBM Redbooks® publication is a follow-on to *Getting started with z/OS Container Extensions and Docker*, SG24-8457, which provides some interesting use cases for zCX.

We start with a brief overview of IBM zCX. In Part 1, “Integration” on page 9, we demonstrate use cases that integrate with zCX. In Part 2, “DevOps in zCX” on page 165, we describe how organizations can benefit from running a DevOps flow in zCX and we describe the set up of necessary components. Finally, in Part 3, “Monitoring and managing zCX systems” on page 229, we discuss IBM Service Management Unite Automation, a free-of-charge customizable dashboard interface and an important discussion of creating the suitable container restart policy.

Authors

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

Lydia Parziale is a Project Leader for the IBM Redbooks team in Poughkeepsie, New York, with domestic and international experience in technology management including software development, project leadership, and strategic planning. Her areas of expertise include business development and database management technologies. Lydia is a PMI certified PMP and an IBM Certified IT Specialist with an MBA in Technology Management and has been employed by IBM for over 25 years in various technology areas.

Marco Egli is a Mainframe Engineer at Swiss Re, based in Switzerland. He worked for more than 10 years in the mainframe area and has been responsible for software and hardware installations and the overall architecture of the mainframe environment. His focus shifted this year towards the use of new technologies on the mainframe.

Maike Havemann is a Client Technical Specialist for systems hardware in Berlin, Germany. She has more than 5 years of experience in the IT industry and within IBM. Her areas of expertise include IBM Z and modern technologies in the field of AI and cloud services.

Subhajit Maitra is a Consulting IT Specialist and member of the IBM America's System Z Hybrid Cloud Technical Sales team based in Hartford, CT. In addition to zCX and Red Hat OpenShift on System Z, his expertise includes IBM App Connect Enterprise, IBM Operational Decision Manager, IBM Integration Bus, IBM MQ, IBM Event Streams, IBM API Connect®, IBM DB2®, and CICS® on Systems Z. Subhajit, who has worked in IT for 27 years, has been an IBM Global Technical Ambassador for Central and Eastern Europe, where he helped IBM clients implement business-critical solutions on IBM Z servers. He has written IBM Redbooks publications about several IBM Middleware, CICS, and DevOps. He holds a master's degree in computer science from Jadavpur University, in Kolkata, India, and is an IBM zChampion.

Eric Marins is a Senior IT Architect in Brazil, focused on hybrid cloud solutions, and infrastructure and platform solutions and competencies, including high availability, Disaster Recovery, networking, Linux, and cloud. Eric works with CIO Private Cloud designing and implementing complex hybrid cloud solutions that involve Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) capabilities. Eric has co-authored several IBM Redbooks publications, including *Advanced Networking Concepts Applied Using Linux on IBM System z*, SG24-7995; *Security for Linux on System z*, SG24-7728; *Scale up for Linux on IBM Z*, REDP-5461; *Getting Started with Docker Enterprise Edition on IBM Z*, SG24-8429, and *Getting started with z/OS Container Extensions and Docker*, SG24-8457.

Edward McCarthy is an IBM certified specialist with over 18 years' experience working with the IBM WebSphere® Application Server on various operating systems, such as z/OS, Linux on IBM Z, AIX®, and Microsoft Windows. He has designed and configured numerous WebSphere Application Server environments for many customers. He also has been involved with all aspects of WebSphere Application Server such as tuning, automating administration, problem solving, and integration. Before joining IBM in 2000, he was a CICS and MQ systems programmer with an Australian government department for over nine years. Edward has worked on several other IBM Redbooks publications and presented on WebSphere topics at various conferences.

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

Robert Haimowitz
IBM Redbooks, Poughkeepsie Center

Gus Kassimis, Ahilan Rajadeva, Yüksel Günal, Joe Bostian, Anthony Giorgio, Gary Puchkoff, Mike Kasper, Kathryn Voss, Peter Spera
IBM USA

Matthew Leming, Mayur Raja, Katherine(Kate) Stanley, David Coles, Alison Lucas, Tony Sharkey
IBM UK

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 ourbooks to be as helpful as possible. Send us your comments about thisbook 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>



IBM z/OS Container Extensions overview

Enterprises increasingly embrace the use of hybrid workloads. Accordingly, the z/OS software system must expand and enable cloud-native workloads on z/OS. IBM z/OS Container Extensions (zCX) is a new feature of z/OS V2R4 that helps to facilitate this expansion.

zCX enables clients to deploy Linux on Z applications as Docker containers in a z/OS system to directly support workloads that have an affinity to z/OS. This process is done without the need to provision a separate Linux server. At the same time, operational control is maintained within z/OS and benefits of z/OS Qualities of Service (QoS) are retained.

Linux on Z applications can run on z/OS, so you can use z/OS operations staff and reuse the z/OS environment.

zCX expands and modernizes the software system for z/OS by including Linux on IBM Z applications, many of which were previously available on only a separately provisioned Linux instance. Most applications (including Systems Management components and development utilities and tools) that are available to run on only Linux can run on z/OS as Docker containers.

Most container images with s390x architecture (the IBM Z opcode set) in the Docker Hub can be run in z/OS Container Extensions. The code is binary-compatible between Linux on IBM Z and z/OS Container Extensions. Also, multi-platform Docker images can run within z/OS Container Extensions.

In addition to open source packages, IBM and third-party software is now available for z/OS 2.4. Clients can now participate with their own Linux applications, which can easily be packaged in Docker format and deployed in the same as open source, IBM, and vendor packages.

This chapter includes the following topics:

- ▶ 1.1, “Why use IBM z/OS Container Extensions?” on page 2
- ▶ 1.2, “IBM zCX architecture” on page 2
- ▶ 1.3, “zCX updates” on page 3

1.1 Why use IBM z/OS Container Extensions?

Today's enterprises increasingly deploy hybrid workloads. In this context, it is critical to embrace agile development practices, and leverage open source packages, Linux applications, and IBM and third-party software packages alongside z/OS applications and data. In today's hybrid world, packaging software as container images is a best practice. zCX gives enterprises the best of both worlds because with zCX, you run Linux on Z applications (which are packaged as Docker containers) on z/OS alongside traditional z/OS workloads.

Consider the use of zCX for the following reasons:

- ▶ You can integrate zCX workloads into your z/OS workload management, infrastructure, and operations strategy. Thus, you take advantage of z/OS strengths, such as pervasive encryption, networking, high availability (HA), and Disaster Recovery (DR).
- ▶ You can design and deploy a hybrid solution that consists of z/OS software and Linux on Z Docker containers on the same z/OS system.
- ▶ You enjoy more open access to data analytics on z/OS by providing developers with standard OpenAPI-compliant RESTful services.

In this way, clients take advantage of z/OS QoS, colocation of applications and data, integrated DR and planned outage coordination, improved resilience, security, and availability.

1.2 IBM zCX architecture

zCX is a feature of z/OS V2R4 that provides a pre-packaged turn-key Docker environment that includes Linux and Docker Engine components that are supported directly by IBM. The initial focus is on base Docker capabilities. zCX workloads are zIIP eligible and provide competitive price performance.

There is limited visibility into the Linux environment. Access is as defined by Docker interfaces. Also, there is little Linux administrative overhead.

The basic architecture for zCX is shown in Figure 1-1. zCX runs as an address space on z/OS, which contains a Linux Kernel, Docker Engine, and the containers that run within that instance.

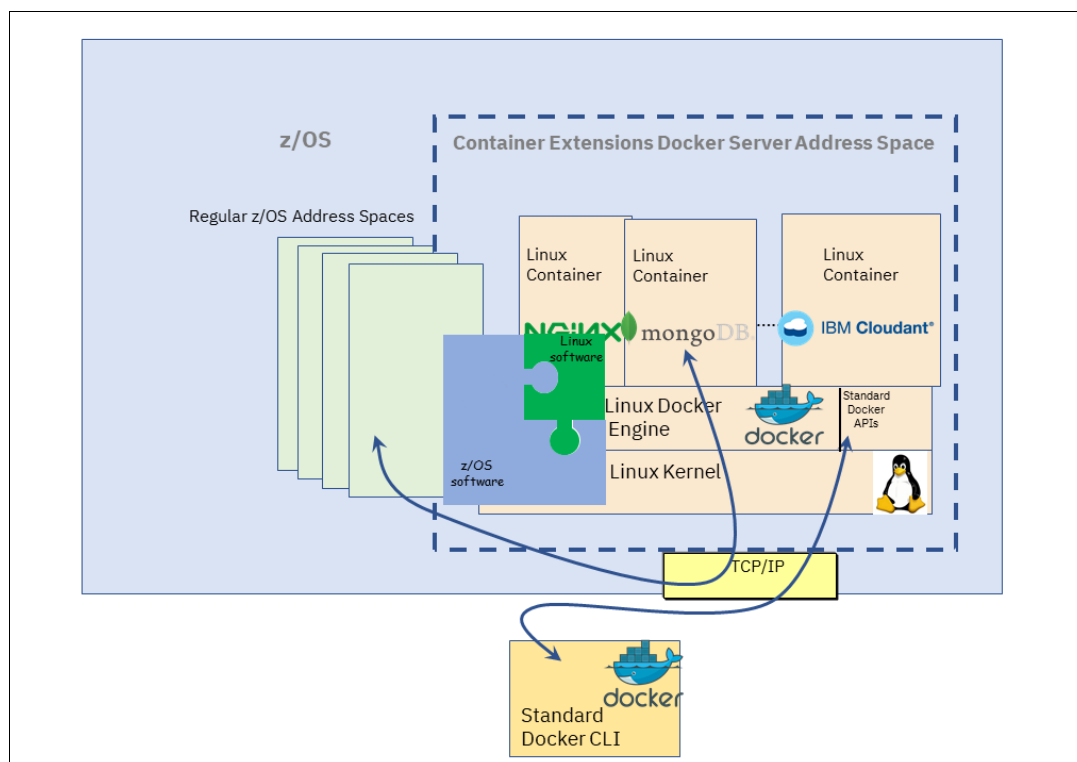


Figure 1-1 zCX basic architecture

Within the zCX address space is the z/OS Linux Virtualization Layer, which enables virtual access to z/OS Storage and Networking. It uses the virtio interface to communicate with the Linux kernel, which enables zCX to support unmodified, open source Linux on Z kernels.

For more information about the zCX architecture, see *Getting started with z/OS Container Extensions and Docker*, SG24-8457.

1.3 zCX updates

Since *Getting started with z/OS Container Extensions and Docker*, SG24-8457, was published, some maintenance in the way of fixes and improvements was released. In this section, we describe some of those recently released maintenance. For more general information about updates on the content that is related to zCX, see [IBM Knowledge Center](#).

For more information about all available fixes that are related to IBM zCX, see this IBM Support [web page](#).

1.3.1 zCX 90-day trial

The feature that is listed in Table 1-1 became available in March 2020.

Table 1-1 90-day trial

APAR	PTF	Description
OA58969	UJ02411	New Function to provide zCX Trial support. Customers can use zCX for a 90-day trial period. This trial period does not require the purchase of the Container Hosting Foundation (feature code 0104). For more information, see: https://www.ibm.com/support/pages/apar/OA58969

1.3.2 Docker proxy certificate support

Table 1-2 provides links to support for Docker proxy certificates along with a description of the support function.

Table 1-2 Docker proxy certificate support

APAR	PTF	Description
OA58236	UJ01141	TRSQ (4) DEFECTS/CORRECTIONS IN ZCX DOCKER GUEST PART 1: <ul style="list-style-type: none">▶ The zCX connect message in the zCX instance joblog is delayed until the zCX Docker CLI container is running.▶ Support for proxy information for the Docker daemon is added. Proxy information is provided by way of the zCX provision or reconfigure workflows. HTTPS proxy servers that are protected by a common CA certificate are supported. HTTPS proxy servers that are protected by a private CA certificate are not supported.▶ Support for manually setting a time zone in the zCX Docker CLI container is added.▶ Support is updated to better track captured debug data so that debug data is not processed multiple times.▶ Support is added to set the zCX Docker CLI hostname to the hostname that is provided on the zCX provision workflow or zCX reconfigure workflow.▶ Support is added to persist zCX Docker CLI user definitions across reconfigure and upgrade workflows.▶ A memory leak in the zCX memory monitor caused the restart. Disabled the memory monitor until a redesign can occur. Docker proxy certificates: https://www.ibm.com/support/pages/apar/OA58236
OA58237	UJ01142	TRSQ (4) DEFECTS/CORRECTIONS IN ZCX DOCKER GUEST PART 2 Docker proxy certificates: https://www.ibm.com/support/pages/apar/OA58237
OA58238	UJ01143	TRSQ (4) DEFECTS/CORRECTIONS IN ZCX DOCKER GUEST PART 3 Docker proxy certificates: https://www.ibm.com/support/pages/apar/OA58238

APAR	PTF	Description
OA58267	UJ01146	<p>Following fixes and new functions are provided with this APAR:</p> <ul style="list-style-type: none"> ▶ Docker proxy configuration support by way of provision and reconfigure workflows ▶ ZCX_DOCKER_ADMIN_SSH_KEYS renamed to ZCX_DOCKER_ADMIN_SSH_KEY ▶ Leading slashes are required for workflow variables path/file input ▶ Trailing slashes are not allowed for workflow variables path/file input ▶ Maximum VSAM LDS allocation size is 46000 MB and enforced by workflows <p>Docker proxy certificates: https://www.ibm.com/support/pages/apar/OA58267</p>
OA58131	UJ01108	<p>zCX is upgrading the level of Linux that is being used in the appliance. This particular upgrade required a small change in the zCX hypervisor code. This hypervisor code was adjusted to continue to work with the current appliance as well as the new level of appliance when it is applied to the system.</p> <p>The GLZ procedure that is used with the START command to instantiate the zCX server was to include a TIME=NOLIMIT. We believe this is a more appropriate default. However, the installation can adjust this as needed.</p> <p>The zCX proc can be handled like any other proc on the system. zCX STOP processing was extended to allow the operator to issue the modify commands while the instance is stopping. This can be used to collect possible failure information while the appliance is attempting to perform the STOP.</p> <p>Docker proxy certificates: https://www.ibm.com/support/pages/apar/OA58131</p>

1.3.3 IBM License Manager Tool support

Table 1-3 lists links to support for the IBM License Manager Tool along with a description of the support function.

Table 1-3 IBM License Manager Tool support

APAR	PTF	Description
OA58598	UJ01574	<p>Add ILMT support: https://www.ibm.com/support/pages/apar/OA58598</p>
OA58599	UJ01575	<p>zCX Server ILMT support part 2: https://www.ibm.com/support/pages/apar/OA58599</p>
OA58600	UJ01577	<p>zCX Server ILMT support part 3: https://www.ibm.com/support/pages/apar/OA58600</p>
OA58621	UJ01576	<p>ILMT disconnected scanner enablement support in zCX provisioning and reconfiguration workflows: https://www.ibm.com/support/pages/apar/OA58621</p>

APAR	PTF	Description
OA58587	UJ01572	<ul style="list-style-type: none"> ► Enables zCX to capture capacity information in support of licensing metrics for containers that are running in the zCX hypervisor. Updates to GLZ Messages: GLZB005I z/OS system service <service> failed, RC=<return_code> RSN=<reason_code> <optional_failure_explanation> ► Added value: Optional_failure_explanation: Unable to obtain LPAR information: https://www.ibm.com/support/pages/apar/OA58587
OA58601	UJ01571	Add ILMT support: https://www.ibm.com/support/pages/apar/OA58601

1.3.4 Performance Improvements (zCX and z/OS Communication Server)

Table 1-4 lists the maintenance features that can provide performance improvements.

Table 1-4 Performance-related improvements

APAR	PTF	Description
OA58296	UJ02045	zCX is adjusting various internal processing to provide improved performance. This is achieved by adjusting latch handling, adjusting timing algorithms, and refactoring code: https://www.ibm.com/support/pages/apar/OA58296
OA58300	UJ01511	This APAR includes enhancements to z/OS CommServer to support zCX workloads. Highlights of the changes include: <ul style="list-style-type: none"> ► Enhancements to support Inbound Workload Queueing (IWQ) for IBM z/OS Container Extensions (zCX) workloads for OSA-Express in QDIO mode. ► Enhancements to offload zCX network processing to zIIPs ► zCX blocking/batching of work elements for more efficient processing of zCX traffic https://www.ibm.com/support/pages/apar/OA58300
PH16581	UI66733	This APAR includes enhancements to z/OS CommServer to support zCX workloads. Highlights of the changes include: <ul style="list-style-type: none"> ► Enhancements to support Inbound Workload Queueing (IWQ) for IBM z/OS Container Extensions (zCX) workloads for OSA-Express in QDIO mode. ► Enhancements to offload zCX network processing to zIIPs ► zCX blocking/batching of work elements for more efficient processing of zCX traffic https://www.ibm.com/support/pages/apar/PH16581

1.3.5 Private CA certificate support for Docker proxy

Table 1-5 lists the improvements that you use to define the CA certificate if a proxy is used to connect by way of HTTPS.

Table 1-5 Private CA certificate support

APAR	PTF	Description
OA58698	UJ02348	Provide private CA certificate support for docker proxy configuration in IBM z/OS Container Extensions (IBM zCX): https://www.ibm.com/support/pages/apar/OA58698
OA58936	UJ02326	zCX allows the specification of an HTTPS proxy server for Docker, but does not allow the user to provide the needed private CA certificate: https://www.ibm.com/support/pages/apar/OA58936
OA58949	UJ02327	Refer to OA58936: https://www.ibm.com/support/pages/apar/OA58949
OA58950	UJ02328	Refer to OA58936: https://www.ibm.com/support/pages/apar/OA58950

1.3.6 Vector / SIMD Support

The APAR that is listed in Table 1-6 allows the zCX instance to use the latest hardware instructions for Single-Instruction, Multiple-Data (SIMD). Unlike instructions that perform a single operation on a single data point, SIMD instructions can perform the same operation on multiple data points. For more information, see *SIMD Business Analytics Acceleration on z Systems*, [REDP-5145](#).

This maintenance involves Vector and SIMD support and makes better use of hardware instructions.

Table 1-6 Vector/SIMD support

APAR	PTF	Description
OA59111	UJ03302	AZIF0144E An unexpected error occurred in LPAR xxxxxx GLZM009I zCX instance xxxx stored failure data: https://www.ibm.com/support/pages/apar/OA59111



Part 1

Integration

In this part, we demonstrate use cases that can use zCX.

This part includes the following chapters:

- ▶ Chapter 2, “Apache Kafka and ZooKeeper” on page 11
- ▶ Chapter 3, “IBM App Connect Enterprise” on page 103
- ▶ Chapter 4, “IBM Aspera fasp.io Gateway” on page 129
- ▶ Chapter 5, “Using IBM MQ on zCX as a client concentrator” on page 147

These chapters demonstrate the capabilities of zCX when used for high-performance coordination, communication between applications, fast file transfer and streaming solutions, and seamless flow of multiple types of data between heterogeneous systems.



Apache Kafka and ZooKeeper

Apache Kafka is an open source, publish-and-subscribe messaging system that is built for high throughput, speed, availability, and scalability. It is an event-streaming software platform for handling real-time data feeds.

Today, billions of data sources continuously generate streams of data records, including streams of events. An event is a digital record of an action that occurred and the time that it occurred. Typically, an event is an action that drives another action as part of a process. A customer placing an order, choosing a seat on a flight, or submitting a registration form are all examples of events. An event does not have to involve a person; for example, a connected thermostat's report of the temperature at a specific time also is an event.

Kafka has three primary capabilities:

- ▶ It enables applications to publish or subscribe to data or event streams.
- ▶ It stores records accurately (that is, in the order in which they occurred) in a fault-tolerant and durable way.
- ▶ It enables records to be processed in real time (by using a tool, such as Kafka Streams).

Kafka is managed with ZooKeeper, which provides capabilities to coordinate the cluster topology and a consistent file system for configuration information.

ZooKeeper is an open source Apache project that provides a centralized service for providing configuration information, naming, synchronization, and group services over large clusters in distributed systems. The goal is to make these systems easier to manage with improved, more reliable propagation of changes. However, ZooKeeper is meant for use by application developers, rather than by administrators.

Note: Soon, Apache Kafka will not require ZooKeeper because of KIP 500. ZooKeeper is a dependency of Kafka rather than it being an extra technology that is required. For more information, see this [web page](#).

Together, these technologies provide reliability, scalability, durability, and performance for many applications.

Typically, a Kafka environment is run on distributed servers because it cannot be run natively on z/OS. By using zCX, you can run Docker containers that host Kafka servers and bring this technology into z/OS.

This chapter provides a step-by-step guide about how to set up and start using Kafka with ZooKeeper for high availability. For a successful installation of Kafka, you must set up ZooKeeper first; therefore, we discuss ZooKeeper first and then Kafka.

This chapter includes the following topics:

- ▶ “ZooKeeper overview” on page 13
- ▶ “ZooKeeper configuration” on page 13
- ▶ “ZooKeeper overview” on page 13
- ▶ “Kafka overview” on page 15
- ▶ “Diagnostic commands” on page 60
- ▶ “Integrating IBM Z applications and Kafka” on page 61
- ▶ “Integration of IBM MQ with Apache Kafka” on page 64
- ▶ “Sending CICS events to Kafka” on page 74

2.1 ZooKeeper overview

If you are running a Kafka cluster, you need centralized management of the entire cluster in terms of leader selection, name service, locking, synchronization, and configuration management. Embedding ZooKeeper means you do not have to build synchronization services from scratch and reinvent services. Kafka uses ZooKeeper to maintain all the cluster metadata including topics, leader, and followers in ZooKeeper.

ZooKeeper provides an infrastructure for cross-node synchronization by maintaining status type information in memory on ZooKeeper instances. A ZooKeeper server keeps a copy of the state of the entire system and persists this information in local log files. Large Kafka clusters are supported by multiple ZooKeeper servers, with a controller server synchronizing the top-level servers.

Within ZooKeeper, Kafka can create what is called a *znode* (ZooKeeper data nodes), which is a file that persists in memory on the ZooKeeper servers. The znode can be updated by any node in the cluster, and any node in the cluster can register to be notified of changes to that znode.

2.2 ZooKeeper configuration

Setting up a Kafka environment requires setting up ZooKeeper first. There is no official pre-built ZooKeeper image available for the s390x platform as of this writing. Therefore, we built a Docker image by using a Dockerfile and instructions that are provided in this section. The benefit of Docker files is that they store the entire procedure on how an image is created.

2.2.1 Persistence with Docker volumes

ZooKeeper must have a persistence layer for its containers in case a container or host fails or is deleted. An environment with a shared persistence layer can be recovered and its state retrieved by restarting the container on another zCX.

zCX offers persistence by way of Docker volumes.

Important: Because persistence is not enabled by default, define and enable them for production use and whenever you want your data to survive any disruptive conditions.

For each ZooKeeper container instance, create two Docker volumes for the following folders:

- /conf - ZooKeeper configuration files
- /data - ZooKeeper data files

For more information about the **docker volume** command, see the [hDocker documentation](#).

The following example includes the **docker volume** command:

```
docker volume create [OPTIONS] [VOLUME]
```

In this chapter, we provide examples of how to create the required Docker volumes as we create them for our examples.

Remember to assign persistent volumes by using the **-v** parameter when you run the **docker run** command to build your ZooKeeper containers.

2.2.2 ZooKeeper default service ports

This section provides information about each of the default ports that are used for a ZooKeeper instance. Table 2-1, we changed the port numbers to make it possible to load multiple instances on the same zCX without conflicts.

Table 2-1 ZooKeeper default ports

Port Number	Description
2181	ZooKeeper clients to connect to the ZooKeeper servers
2888	Peer ZooKeeper servers to communicate with each other
3888	Leader election
8080	AdminServer

You can choose any ports of your liking. It is recommended that you use different ports on the ZooKeeper instances to avoid conflicts in a high availability environment.

Our ZooKeeper cluster consists of three containers: 1, 2, and 3. Each container uses two ports (3888 and 2888) for internal cluster communications and uses port 2128 for clients.

Because some of the containers can be on the same zCX instance, you can add a suffix (container ID) to each port to prevent port collisions. An example on how to define these ports in this way is shown in Table 2-2.

Table 2-2 iZooKeeper cluster ports

Container ID	Container name	Ports
1	zk-app-1	12181, 12888, 13888, 18080
2	zk-app-2	22181, 22888, 23888, 28080
3	zk-app-3	32181, 32888, 33888, 38080

During container creation (**docker run**), we do not use the AdminServer port (8080) because of security reasons. The administration can be performed by running the **curl** and **wget** commands inside the container.

2.2.3 ZooKeeper configuration file

The `/conf/zoo.cfg` file contains the configuration definition for the ZooKeeper instance. The following parameters must be updated:

- **tickTime**: Specifies the duration (in milliseconds) at which ZooKeeper checks the status of the hosts; for example:
`tickTime=2000`
- **dataDir**: Specifies the directory to store the in-memory database. If this directory does not exist, create it and ensure that the user has read/write permissions; for example:
`dataDir=/data`
- **clientPort**: Specifies the port that the ZooKeeper client listens on for connections; for example:
`clientPort=2181`

- ▶ `maxClientCnxns`: Limits the maximum number of client connections; for example:
`maxClientCnxns=60`
- ▶ `server.n`: (Optional) Specifies host names and ports for management servers in order of failover if replicated servers are used, where `n` identifies the main management server, followed by other servers in the order of priority for failover; for example:
`server.1=dbhost1:2888:3888`
`server.2=dbhost2:2888:3888`

A `tickTime` of 2000 milliseconds is the suggested interval between heartbeats. A shorter interval might lead to system overhead with limited benefits. The `dataDir` parameter points to the `/data`.

Typically, ZooKeeper uses port 2181 to listen for client connections. However, we use a different port number to identify the container instance. In most situations, 60 allowed client connections are plenty for development and testing. You can increase this setting for production systems.

For more information about ZooKeeper settings, see this [web page](#).

2.3 Kafka overview

The Kafka cluster stores streams of records in categories called *topics*. Kafka topics are feeds of messages in categories. Processes that publish messages to a topic are called *producers*. Processes that subscribe to topics and process the feed of published messages are called *consumers*. For more information and examples of producers and consumers, see this [web page](#).

You can see all messages from all suite products, and you can use these messages with any Kafka client implementation. Any application that works with any type of data (logs, events, and more), and requires that data is transferred can benefit from Kafka.

A consumer can direct messages to a window. You can use the consumer to see these messages. Kafka uses the ZooKeeper API, which is a centralized service that maintains configuration information.

Kafka is used primarily for creating two kinds of applications:

- ▶ Real-time streaming data pipelines
These applications are designed specifically to move millions and millions of data or event records between enterprise systems (at scale and in real time) and move them reliably, without risk of corruption, duplication of data, and other problems that typically occur when moving such huge volumes of data at high speeds.
- ▶ Real-time streaming applications
These applications are driven by record or event streams and that generate streams of their own. If you spend any time online, you encounter scores of these applications every day, from the retail site that continually updates the quantity of a product at your local store, to sites that display personalized recommendations or advertising based on click stream analysis.

Also, Kafka is an especially capable solution whenever you are dealing with large volumes of data and require real-time processing to make that data available to others. Kafka has the following key capabilities:

- Publish and subscribe streams of records. Data is stored so that applications can pull the information that they need, and track what they saw thus far.
- It can handle hundreds of read and write operations per second from many producers and consumers
- Atomic broadcast, send a record once, every subscriber gets it once.
- Store streams of data records on disk and replicate within the distributed cluster for fault-tolerance. Persist data for a specific period before deleting.
- Built on top of the ZooKeeper synchronization service to keep topic, partitions, and metadata highly available.

Figure 2-1 shows Kafka's key components.

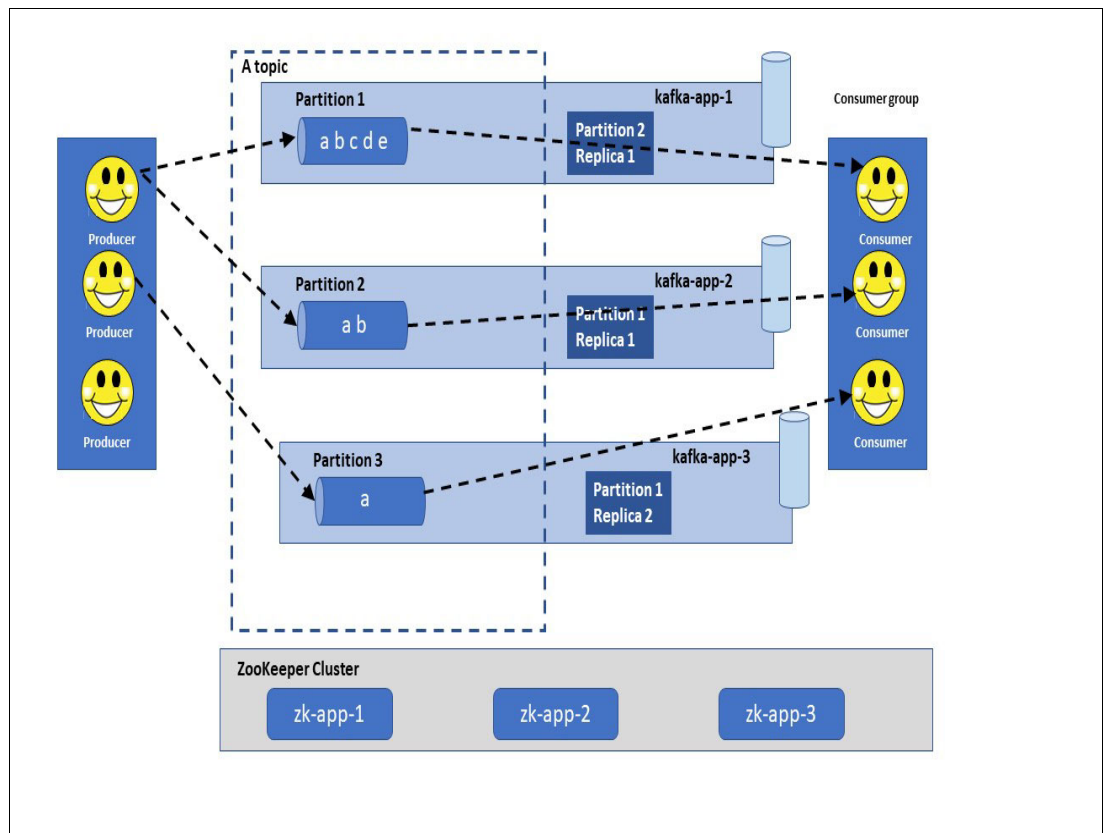


Figure 2-1 Kafka key components

2.4 Kafka configuration

In this section, we describe configuring a Kafka cluster that consists of three servers that are running in Docker containers on zCX. We also provide the steps that are required to set up our cluster environment on zCX.

In our environment, we defined the three servers in one zCX instance. This configuration means that we must assign a different TCPIP port to each one, as listed in Table 2-3.

Table 2-3 iKafka ports

Container ID	Container name	Ports
#1	kafka-app-1	19092
#2	kafka-app-2	29092
#3	kafka-app-3	39092

2.4.1 Persistence with Docker volumes

For each Kafka container instance, you must create a Docker volume that stores the following folder:

/home/kafka/config - Kafka configuration files
/home/kafka/logs/ - Kafka log files

For more information about the **docker volume** command, see the Docker documentation that is available at this [web page](#).

The use of the **docker volume** command is shown in the following example:

```
docker volume create [OPTIONS] [VOLUME]
```

Remember to assign persistent volumes by using the **-v** parameter when you run the **docker run** command to build your Kafka containers.

2.4.2 Kafka configuration file

The /home/kafka/config/server.properties file contains the configuration definition for the Kafka instance and should have the following parameters updated:

- ▶ **broker.id**: The ID of the broker. This ID must be set to a unique integer for each broker.
- ▶ **listeners**: The address on which the socket server listens.
- ▶ **advertised.listeners**: The host name and port the broker advertises to producers and consumers.
- ▶ **zookeeper.connect**: ZooKeeper connection string.

For more information about Kafka parameters, see this [web page](#).

2.5 Kafka configuration on one zCX instance

After you review 2.4.1, “Persistence with Docker volumes” on page 17 and section 2.4.2, “Kafka configuration file” on page 17, you continue with creating a Kafka cluster. At this point, you must decide whether to configure Kafka on one zCX instance, as described in this section, or on multiple zCX instances, as described in 2.6, “Kafka configuration on three zCX instances” on page 39.

2.5.1 Creating the Kafka cluster

Figure 2-2 and Table 2-4 on page 19 show a zCX use case in our IBM Redbooks environment. The IBM Z machines that we used for this lab are IBM z15™.

This section describes how to set up and deploy a Kafka cluster on one zCX instance that has three nodes for each application (three ZooKeeper and three Kafka).

The Docker container names that we use for this example are zk-app-1, zk-app-2, and zk-app-3 for ZooKeeper and kafka-app-1, kafka-app-2, and kafka-app-3 for Kafka. We appended the container instance number to help to identify each instance. Also, this number is appended to the port number to help to allow all containers to run in one zCX instance, as shown in Figure 2-2.

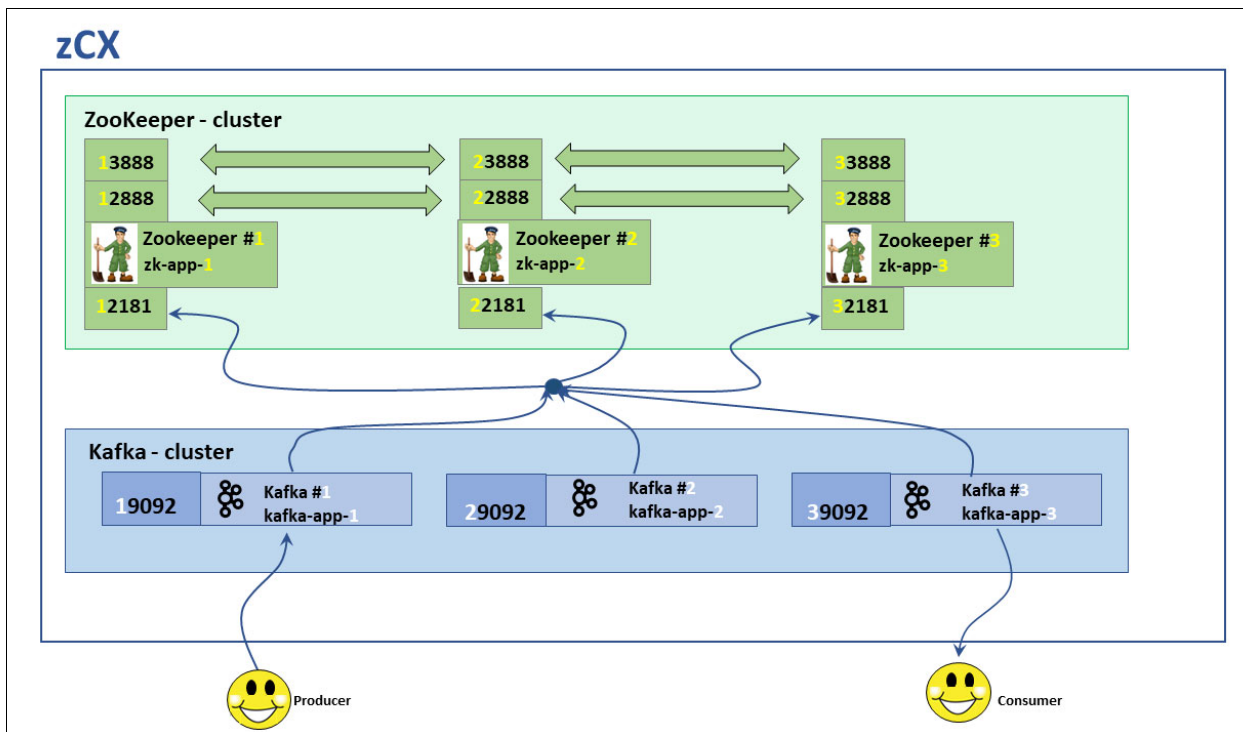


Figure 2-2 Kafka cluster diagram

Table 2-4 lists our Kafka cluster information.

Table 2-4 Kafka cluster information

Container name	Instance number	Application	Persistent Volumes (PV)	Application ports
zk-app-1	1	ZooKeeper	zoo1-conf:/conf zoo1-data:/data	12181, 12888, 13888
zk-app-2	2	ZooKeeper	zoo2-conf:/conf zoo2-data:/data	22181, 22888, 23888
zk-app-3	3	ZooKeeper	zoo3-conf:/conf zoo3-data:/data	32181, 32888, 33888
kafka-app-1	1	Kafka	kafka1-conf:/home/kafka/config kafka1-logs:/home/kafka/logs	19092
kafka-app-2	2	Kafka	kafka2-conf:/home/kafka/config kafka2-logs:/home/kafka/logs	29092
kafka-app-3	3	Kafka	kafka3-conf:/home/kafka/config kafka3-logs:/home/kafka/logs	39092

Notice that the container names, persistent volumes, and application ports have the instance number appended to them to support the running of all containers in a single zCX host. This configuration also can be done when multiple zCX instances are used.

Important: Be aware that setting up multiple servers on a single machine does not create any redundancy and your environment is vulnerable if a host failure occurs. If something occurs that caused the machine to stop, all of the ZooKeeper servers go offline. Full redundancy requires that each container have its own zCX instance and on separate pieces of hardware.

We deploy ZooKeeper and Kafka containers to `sc74cn09.pbm.ihost.com` (129.40.23.76). We completed the steps that are described in the next section, “Creating ZooKeeper file”, while connected to our system that is named `sc74cn09`.

Creating ZooKeeper file

No official ZooKeeper Docker image is available on the Docker hub as of this writing. The steps that are described in this section build a ZooKeeper image to run on the IBM Z platform. For more information about the latest Dockerfile examples for ZooKeeper, see this [web page](#).

Complete the following steps to create the ZooKeeper file:

1. Example 2-1 shows creating a folder and downloading the two required files from GitHub.

Example 2-1 Creating the ZooKeeper file

```
mkdir -p /home/admin/zookeeper

cd zookeeper

curl -k -o Dockerfile.zookeeper
https://raw.githubusercontent.com/31z4/zookeeper-docker/master/3.6.1/Dockerfile
```

```
curl -k -o docker-entripoint.sh
https://raw.githubusercontent.com/31z4/zookeeper-docker/master/3.6.1/docker-entripoint.sh
```

```
chmod 755 docker-entripoint.sh
```

Note: If problems occur when you use the curl commands to download the required files, download these files from a Linux server and then, upload to your zCX instance by using the **scp** or **ssh** commands. Complete the following steps:

- a. Download the Dockerfile to /tmp/ and name it as Dockerfile.zookeeper.
- b. Download docker-entripoint.sh to /tmp/.
- c. Run the following commands to upload:


```
cat /tmp/Dockerfile.zookeeper | ssh -p 8022 admin@<zcx IP address> "cat
> /home/admin/Dockerfile.zookeeper"

cat /tmp/docker-entripoint.sh | ssh -p 8022 admin@<zcx IP address> "cat
> /home/admin/Docker-entripoint.sh "
```

The Dockerfile.zookeeper file is written to: /home/admin/Dockerfile.zookeeper.

The docker-entripoint.sh file is written to: /home/admin/docker-entripoint.sh.

2. Edit the Dockerfile.zookeeper file with your details as shown Example 2-2 and highlighted in **bold**.

Example 2-2 Dockerfile.zookeeper file

FROM s390x/ubuntu:18.04

ENV JAVA_HOME=/usr/lib/jvm/java-11-openjdk-s390x

```
ENV ZOO_CONF_DIR=/conf \
    ZOO_DATA_DIR=/data \
    ZOO_DATA_LOG_DIR=/data/log \
    ZOO_LOG_DIR=/logs \
    ZOO_TICK_TIME=2000 \
    ZOO_INIT_LIMIT=5 \
    ZOO_SYNC_LIMIT=2 \
    ZOO_AUTOPURGE_PURGEINTERVAL=0 \
    ZOO_AUTOPURGE_SNAPRETAINCOUNT=3 \
    ZOO_MAX_CLIENT_CNXNS=60 \
    ZOO_STANDALONE_ENABLED=true \
    ZOO_ADMIN_SERVER_ENABLED=true
```

```
# Add a user with an explicit UID/GID and create necessary directories
```

```
RUN set -eux; \
    groupadd -r zookeeper --gid=1000; \
    useradd -r -g zookeeper --uid=1000 zookeeper; \
    mkdir -p "$ZOO_DATA_LOG_DIR" "$ZOO_DATA_DIR" "$ZOO_CONF_DIR" "$ZOO_LOG_DIR"; \
    chown zookeeper:zookeeper "$ZOO_DATA_LOG_DIR" "$ZOO_DATA_DIR" "$ZOO_CONF_DIR"
"$ZOO_LOG_DIR"
```

```
# Install required packages
```

```
RUN set -eux; \
```

```

apt-get update; \
DEBIAN_FRONTEND=noninteractive \
apt-get install -y --no-install-recommends \
    openjdk-11-jre \
    tar \
    wget \
    vim \
    ca-certificates \
    dirmngr \
    gosu \
    gnupg \
    netcat \
rm -rf /var/lib/apt/lists/*; \
# Verify that gosu binary works
gosu nobody true

ARG GPG_KEY=BBE7232D7991050B54C8EA0ADC08637CA615D22C
ARG SHORT_DISTRO_NAME=zookeeper-3.6.1
ARG DISTRO_NAME=apache-zookeeper-3.6.1-bin

# Download Apache ZooKeeper, verify its PGP signature, untar and clean up
RUN set -eux; \
ddist() { \
    local f="$1"; shift; \
    local distFile="$1"; shift; \
    local success=; \
    local distUrl=; \
    for distUrl in \
        'https://www.apache.org/dyn/closer.cgi?action=download&filename=' \
        https://www-us.apache.org/dist/ \
        https://www.apache.org/dist/ \
        https://archive.apache.org/dist/ \
    ; do \
        if wget -q -O "$f" "$distUrl$distFile" && [ -s "$f" ]; then \
            success=1; \
            break; \
        fi; \
    done; \
    [ -n "$success" ]; \
}; \
ddist "$DISTRO_NAME.tar.gz"
"zookeeper/$SHORT_DISTRO_NAME/$DISTRO_NAME.tar.gz"; \
ddist "$DISTRO_NAME.tar.gz.asc"
"zookeeper/$SHORT_DISTRO_NAME/$DISTRO_NAME.tar.gz.asc"; \
export GNUPGHOME="$(mktemp -d)"; \
gpg --keyserver ha.pool.sks-keyservers.net --recv-key "$GPG_KEY" || \
gpg --keyserver pgp.mit.edu --recv-keys "$GPG_KEY" || \
gpg --keyserver keyserver.pgp.com --recv-keys "$GPG_KEY"; \
gpg --batch --verify "$DISTRO_NAME.tar.gz.asc" "$DISTRO_NAME.tar.gz"; \
tar -zxvf "$DISTRO_NAME.tar.gz"; \
mv "$DISTRO_NAME/conf/*" "$ZOO_CONF_DIR"; \
rm -rf "$GNUPGHOME" "$DISTRO_NAME.tar.gz" "$DISTRO_NAME.tar.gz.asc"; \
chown -R zookeeper:zookeeper "/$DISTRO_NAME"

WORKDIR $DISTRO_NAME

```

```
VOLUME ["$ZOO_DATA_DIR", "$ZOO_DATA_LOG_DIR", "$ZOO_LOG_DIR"]
```

```
#EXPOSE 2181 2888 3888 8080
```

```
ENV PATH=$PATH:$DISTRO_NAME/bin \  
    ZOO_CFGDIR=$ZOO_CONF_DIR
```

```
COPY docker-entrypoint.sh /  
ENTRYPOINT ["/docker-entrypoint.sh"]  
CMD ["zkServer.sh", "start-foreground"]
```

Note: Writing EXPOSE in the Dockerfile is merely a hint that a certain port is useful. Docker does not do anything with that information by itself.

ZooKeeper ports are different for each container instance; therefore, we can safely comment out the EXPOSE 2181 2888 3888 8080 line from this file.

3. Run the following **docker build** command to build the ZooKeeper image:

```
docker build -t zcx-zookeeper-img -f Dockerfile.zookeeper
```

We named the new image zcx-zookeeper-img.

4. Create two persistent volumes for each zookeeper instance by using the following commands:

- For zk-app-1:

```
docker volume create zoo1-conf  
docker volume create zoo1-data
```

- For zk-app-2:

```
docker volume create zoo2-conf  
docker volume create zoo2-data
```

- For zk-app-3:

```
docker volume create zoo3-conf  
docker volume create zoo3-data
```

Complete the following steps to populate the volumes for each container instance. We use a dummy temporary image in our example:

1. For zk-app-1, run the commands that are shown in Example 2-3.

Example 2-3 Populate the volumes for zk-app-1

```
docker run --name dummy -v zoo1-conf:/conf1 -v zoo1-data:/data1 -d zcx-zookeeper-img
```

```
docker exec -it dummy bash
```

```
cp -a /conf/* /conf1/
```

```
cp -a /data/* /data1/
```

```
apt-get update
```

```
apt install -y vim
```

Edit the `/conf1/zoo.cfg` file by running the `vi` command with the following instructions.

- a. Delete the following line:

```
server.1=localhost:2888:3888;2181 line
```

- b. Update the following values:

```
initLimit=10
syncLimit=5
standaloneEnabled=false
```

- c. Add the following lines, changing the IP address to your IP addresses:

```
server.1=0.0.0.0:12888:13888;12181
server.2=129.40.23.76:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181
```

After the updates, the `zoo.cfg` file should look similar to the file that is shown in Example 2-4.

Example 2-4 /conf1/zoo.cfg file

```
dataDir=/data
dataLogDir=/dataLog
tickTime=2000
initLimit=10
syncLimit=5
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
standaloneEnabled=false
admin.enableServer=true
server.1=0.0.0.0:12888:13888;12181
server.2=129.40.23.76:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181
```

- d. Edit `/data1/myid` file by running the `vi` command and update with the number 1 for our first ZooKeeper instance.

- e. Run the `exit` command to leave the dummy container.

- f. Run the `docker rm dummy --force` command to delete the dummy container.

2. For `zk-app-2`, run the commands that are shown in Example 2-5.

Example 2-5 Populate the volumes for zk-app-2

```
docker run --name dummy -v zoo2-conf:/conf2 -v zoo2-data:/data2 -d
zcx-zookeeper-img
```

```
docker exec -it dummy bash
```

```
cp -a /conf/* /conf2/
```

```
cp -a /data/* /data2/
```

```
apt-get update
```

```
apt install -y vim
```

Edit the /conf2/zoo.cfg file by running the **vi** command with the following instructions:

- a. Delete the following line:
server.1=localhost:2888:3888;2181 line
- b. Update the following values:
initLimit=10
syncLimit=5
standaloneEnabled=false
- c. Add the following lines and use your own IP address:
server.1=129.40.23.76:12888:13888;12181
server.2=0.0.0.0:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181

After the updates, zoo.cfg file should look similar to that shown in Example 2-6:

Example 2-6 /conf2/zoo.cfg file

```
dataDir=/data
dataLogDir=/dataLog
tickTime=2000
initLimit=10
syncLimit=5
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
standaloneEnabled=false
admin.enableServer=true
server.1=129.40.23.76:12888:13888;12181
server.2=0.0.0.0:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181
```

- d. Edit the /data2/myid file by running the **vi** command and update with the number 2 for our second ZooKeeper instance.
 - e. Run the **exit** command to leave the dummy container.
 - f. Run the **docker rm dummy --force** command to delete the dummy container.
3. For zk-app-3, run the commands that are shown in Example 2-7.

Example 2-7 Populate the volumes for zk-app-3

```
docker run --name dummy -v zoo3-conf:/conf3 -v zoo3-data:/data3 -d
zcx-zookeeper-img

docker exec -it dummy bash

cp -a /conf/* /conf3/

cp -a /data/* /data3/

apt-get update

apt install -y vim
```

a. Edit the `/conf3/zoo.cfg` file by running the `vi` command:

i. Delete the following line:

```
server.1=localhost:2888:3888;2181 line
```

ii. Update the following values:

```
initLimit=10
syncLimit=5
standaloneEnabled=false
```

iii. Add the following lines and use your IP address:

```
server.1=129.40.23.76:12888:13888;12181
server.2=129.40.23.76:22888:23888;22181
server.3=0.0.0.0:32888:33888;32181
```

After the updates, the `zoo.cfg` file should look similar to Example 2-8.

Example 2-8 /conf3/zoo.cfg file

```
dataDir=/data
dataLogDir=/dataLog
tickTime=2000
initLimit=10
syncLimit=5
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
standaloneEnabled=false
admin.enableServer=true
server.1=129.40.23.76:12888:13888;12181
server.2=129.40.23.76:22888:23888;22181
server.3=0.0.0.0:32888:33888;32181
```

b. Edit the `/data3/myid` file by running the `vi` command and update with the number 3 for our third ZooKeeper instance.

c. Run the `exit` command to leave the dummy container.

d. Run the `docker rm dummy --force` command to delete the dummy container.

4. Start your three ZooKeeper instances. Because ZooKeeper ‘fails fast’, it is better to always restart it. We appended `--restart always` in the Docker run commands, as shown in Example 2-9.

Example 2-9 Docker run commands

```
docker run --name zk-app-1 -p 12181:12181 -p 12888:12888 -p 13888:13888 -v
zoo1-conf:/conf:rw -v zoo1-data:/data:rw --restart always -d zcx-zookeeper-img
```

```
docker run --name zk-app-2 -p 22181:22181 -p 22888:22888 -p 23888:23888 -v
zoo2-conf:/conf:rw -v zoo2-data:/data:rw --restart always -d zcx-zookeeper-img
```

```
docker run --name zk-app-3 -p 32181:32181 -p 32888:32888 -p 33888:33888 -v
zoo3-conf:/conf:rw -v zoo3-data:/data:rw --restart always -d zcx-zookeeper-img
```

5. Run the following commands to verify the startup of each container:

```
docker logs zk-app-1
docker logs zk-app-2
docker logs zk-app-3
```

For each command, verify that you receive a message, as shown in Example 2-10.

Example 2-10 Example messages of successful start of each container

```
2020-06-09 16:41:30,773 [myid:1] - INFO [main:ContextHandler@825] - Started
o.e.j.s.ServletContextHandler@33ecda92{/,null,AVAILABLE}
2020-06-09 16:41:30,778 [myid:1] - INFO [main:AbstractConnector@330] - Started
ServerConnector@4c60d6e9{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
2020-06-09 16:41:30,778 [myid:1] - INFO [main:Server@399] - Started @975ms
2020-06-09 16:41:30,778 [myid:1] - INFO [main:JettyAdminServer@178] - Started
AdminServer on address 0.0.0.0, port 8080 and command URL /commands
```

6. For the zk-app-1 instance, run the following commands to confirm ZooKeeper is operational:

```
docker exec -it zk-app-1 bash -c "wget -O -
http://localhost:8080/commands/stats"
```

You should receive output similar to that shown in Example 2-11. Highlighted in **bold** is the state of each instance. One instance should be a leader and the others followers.

Example 2-11 Confirmation of follower instance

```
admin@16bbba5088fb:~/kafka$ docker exec -it zk-app-1 bash -c "wget -O -
http://localhost:8080/commands/stats"
--2020-06-09 22:13:55-- http://localhost:8080/commands/stats
Resolving localhost (localhost)... 127.0.0.1, ::1
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 923 [application/json]
Saving to: 'STDOUT'
```

```
-          0%[          ]      0  --.-KB/s
{
  "version" : "3.6.1--104dcb3e3fb464b30c5186d229e00af9f332524b, built on
04/21/2020 15:01 GMT",
  "read_only" : false,
  "server_stats" : {
    "packets_sent" : 0,
    "packets_received" : 0,
    "fsync_threshold_exceed_count" : 0,
    "client_response_stats" : {
      "last_buffer_size" : -1,
      "min_buffer_size" : -1,
      "max_buffer_size" : -1
    },
    "uptime" : 19944686,
    "provider_null" : false,
    "server_state" : "follower",
    "outstanding_requests" : 0,
    "min_latency" : 0,
    "avg_latency" : 0.0,
    "max_latency" : 0,
    "data_dir_size" : 67108880,
    "log_dir_size" : 459,
    "last_processed_zxid" : 4294967388,
    "num_alive_client_connections" : 0
  },
}
```

```

"client_response" : {
  "last_buffer_size" : -1,
  "min_buffer_size" : -1,
  "max_buffer_size" : -1
},
"node_count" : 55,
"connections" : [ ],
"secure_connections" : [ ],
"command" : "stats",
"error" : null
-          100%[=====]          923  --.-KB/s    in 0s

2020-06-09 22:13:55 (356 MB/s) - written to stdout [923/923]

```

7. For the zk-app-2 instance, run the following commands to confirm ZooKeeper is operational:

```

docker exec -it zk-app-2 bash -c "wget -O -
http://localhost:8080/commands/stats"

```

You should receive similar output as shown in Example 2-12. Highlighted in **bold** is the state of each instance. One instance should be a leader and the others followers.

Example 2-12 Confirmation of leader instance

```

admin@16bbba5088fb:~/kafka$ docker exec -it zk-app-2 bash -c "wget -O -
http://localhost:8080/commands/stats"
--2020-06-09 22:21:46-- http://localhost:8080/commands/stats
Resolving localhost (localhost)... 127.0.0.1, ::1
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1223 (1.2K) [application/json]
Saving to: 'STDOUT'

```

```

-          0%[
]          0  --.-KB/s    {
  "version" : "3.6.1--104dcb3e3fb464b30c5186d229e00af9f332524b, built on
04/21/2020 15:01 GMT",
  "read_only" : false,
  "server_stats" : {
    "packets_sent" : 24877,
    "packets_received" : 24877,
    "fsync_threshold_exceed_count" : 0,
    "client_response_stats" : {
      "last_buffer_size" : 16,
      "min_buffer_size" : 16,
      "max_buffer_size" : 196
    },
    "provider_null" : false,
    "uptime" : 112174649,
    "server_state" : "leader",
    "outstanding_requests" : 0,
    "min_latency" : 0,
    "avg_latency" : 0.4014,
    "max_latency" : 6009,
    "data_dir_size" : 67108880,
    "log_dir_size" : 459,

```

```

    "last_processed_zxid" : 4294967388,
    "num_alive_client_connections" : 1
  },
  "client_response" : {
    "last_buffer_size" : 16,
    "min_buffer_size" : 16,
    "max_buffer_size" : 196
  },
  "proposal_stats" : {
    "last_buffer_size" : 253,
    "min_buffer_size" : 36,
    "max_buffer_size" : 340
  },
  "node_count" : 55,
  "connections" : [ {
    "remote_socket_address" : "172.17.0.1:48640",
    "interest_ops" : 1,
    "outstanding_requests" : 0,
    "packets_received" : 16518,
    "packets_sent" : 16518
  } ],
  "secure_connections" : [ ],
  "command" : "stats",
  "error" : null
-
100%[=====] 1.19K --.-KB/s
in 0s

```

2020-06-09 22:21:46 (29.1 MB/s) - written to stdout [1223/1223]

8. For the zk-app-3 instance, run the following commands to confirm ZooKeeper is operational:

```

docker exec -it zk-app-3 bash -c "wget -O -
http://localhost:8080/commands/stats"

```

You should receive output shown in Example 2-13. Highlighted in **bold** is the state of each instance. One instance should be a leader and the others followers.

Example 2-13 Confirmation of second follower instance

```

admin@16bbba5088fb:~/kafka$ docker exec -it zk-app-3 bash -c "wget -O -
http://localhost:8080/commands/stats"
--2020-06-09 22:23:27-- http://localhost:8080/commands/stats
Resolving localhost (localhost)... 127.0.0.1, ::1
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1274 (1.2K) [application/json]
Saving to: 'STDOUT'

-
0%[          ] 0 --.-KB/s
{
  "version" : "3.6.1--104dcb3e3fb464b30c5186d229e00af9f332524b, built on
04/21/2020 15:01 GMT",
  "read_only" : false,
  "server_stats" : {
    "packets_sent" : 9858,

```

```

    "packets_received" : 9858,
    "fsync_threshold_exceed_count" : 0,
    "client_response_stats" : {
      "last_buffer_size" : 16,
      "min_buffer_size" : 16,
      "max_buffer_size" : 16
    },
    "provider_null" : false,
    "uptime" : 112260324,
    "server_state" : "follower",
    "outstanding_requests" : 0,
    "min_latency" : 0,
    "avg_latency" : 0.1301,
    "max_latency" : 32,
    "data_dir_size" : 67108880,
    "log_dir_size" : 1100,
    "last_processed_zxid" : 4294967388,
    "num_alive_client_connections" : 2
  },
  "client_response" : {
    "last_buffer_size" : 16,
    "min_buffer_size" : 16,
    "max_buffer_size" : 16
  },
  "node_count" : 55,
  "connections" : [ {
    "remote_socket_address" : "172.17.0.1:35474",
    "interest_ops" : 1,
    "outstanding_requests" : 0,
    "packets_received" : 6422,
    "packets_sent" : 6422
  }, {
    "remote_socket_address" : "172.17.0.1:35508",
    "interest_ops" : 1,
    "outstanding_requests" : 0,
    "packets_received" : 3436,
    "packets_sent" : 3436
  } ],
  "secure_connections" : [ ],
  "command" : "stats",
  "error" : null
-
100%[=====>] 1.24K --.-KB/s in 0s

```

2020-06-09 22:23:27 (567 MB/s) - written to stdout [1274/1274]

The ZooKeeper installation is completed. Next, you install Kafka.

Kafka Docker images

In this section, we provide the steps to create images to install Kafka. For more information about the most current Dockerfile examples for Kafka, see this [web page](#).

Complete the following steps:

1. Run the following commands to create a folder for the Kafka image:

```
mkdir /home/admin/kafka
cd /home/admin/kafka
```

2. Download the Kafka Dockerfile by running the following command:

```
curl -k -o Dockerfile.kafka
https://raw.githubusercontent.com/linux-on-ibm-z/dockerfile-examples/master/ApacheKafka/Dockerfile
```

Note: If you encounter problems when the **curl** commands are used to download the required files, download these files from a Linux server and then upload to your zCX instance by running the **scp** or **ssh** commands. Complete the following steps:

- a. Download the Dockerfile into /tmp/ and name it as Dockerfile.kafka.
- b. Download docker-entrypoint.sh into /tmp/.
- c. Run the following commands to upload:

```
cat /tmp/Dockerfile.kafka | ssh -p 8022 admin@<zcx IP address> "cat >
/home/admin/Dockerfile.kafka"
```

The Dockerfile.kafka file is written to /home/admin/Dockerfile.kafka.

3. Edit the file Dockerfile.kafka to look similar to that shown in Example 2-14. Highlighted in **bold** is what is required to update, add or remove.

Example 2-14 Dockerfile.kafka

```
##### dockerfile for Apache Kafka version 2.5.0 #####
#
# This Dockerfile builds a basic installation of Apache Kafka.
#
# Kafka is run as a cluster on one or more servers. The Kafka cluster stores
# streams of records in categories called topics.
# Each record consists of a key, a value, and a timestamp.
# In Kafka the communication between the clients and the servers is done with a
# simple, high-performance, language agnostic TCP protocol.
#
# To build this image, from the directory containing this Dockerfile
# (assuming that the file is named Dockerfile):
# docker build -t <image_name> .
#
# To Start Apache Kafka run the below command:
# docker run --name <container_name> -d <image>
#
# To check Apache kafka is running, Enter below command:
# docker exec <container_id of kafka> <any kafka related command>
# Eg. To list topic and message files:
# docker exec <container_id of kafka> bin/kafka-topics.sh --list --zookeeper
localhost:2181
#
# Reference:
# http://kafka.apache.org/
# https://kafka.apache.org/quickstart
#
```



```
#####
###

# Base Image
FROM s390x/ubuntu:16.04

# The author
LABEL maintainer="LoZ Open Source Ecosystem
(https://www.ibm.com/developerworks/community/groups/community/lozopensource)"

ENV SOURCE_DIR=/home/
ENV JAVA_HOME=/home/jdk-11.0.5+10
ENV PATH=$PATH:$SOURCE_DIR:$JAVA_HOME/bin
ENV VERSION=2.12-2.5.0

WORKDIR $SOURCE_DIR

# Install dependencies
RUN apt-get update && apt-get -y install \
git \
unzip \
wget \
vim \
net-tools \
kafkacat \
# Download Adopt JDK
&& wget
https://github.com/AdoptOpenJDK/openjdk11-binaries/releases/download/jdk-11.0.5
%2B10/OpenJDK11U-jdk_s390x_linux_hotspot_11.0.5_10.tar.gz \
&& tar -xvzf OpenJDK11U-jdk_s390x_linux_hotspot_11.0.5_10.tar.gz \
&& rm OpenJDK11U-jdk_s390x_linux_hotspot_11.0.5_10.tar.gz \
# Download the Apache Kafka Binary
&& wget http://mirrors.estointernet.in/apache/kafka/2.5.0/kafka_${VERSION}.tgz
\
&& tar -xvzf kafka_${VERSION}.tgz \
&& rm kafka_${VERSION}.tgz \
&& mv kafka_${VERSION} kafka
# Expose ports for Apache ZooKeeper and kafka
# EXPOSE 2181 9092

WORKDIR $SOURCE_DIR/kafka/

# start zookeeper and kafka server
CMD bin/kafka-server-start.sh config/server.properties

# End of Dockerfile
```

Note: Writing EXPOSE in Dockerfile is merely a hint that a specific port is useful. Docker does not use that information.

4. To build a new Kafka image and tag it as zcx-kafka-img, run the following command:
`docker build -t zcx-kafka-img -f Dockerfile.kafka`

You receive the following messages if the build completes successfully:

```
Successfully built 3ac1c714ee08
Successfully tagged zcx-kafka-img:latest
```

5. Define Kafka volume groups for each Kafka container instance and logs by running the following commands:

```
docker volume create kafka1-conf
docker volume create kafka1-logs
docker volume create kafka2-conf
docker volume create kafka2-logs
docker volume create kafka3-conf
docker volume create kafka3-logs
```

Running these commands makes the configuration files persistent for Kafka, which allows your system to survive during an upgrade. For more information about persistent volumes, see *Getting started with z/OS Container Extensions and Docker*, SG24-8457.

Complete the following steps to create the configuration file for each Kafka container image:

1. For kafka-app-1, run the commands shown in Example 2-15.

Example 2-15 Create the configuration file for kafka-app1

```
docker run --name dummy -it --entrypoint /bin/bash -v
kafka1-conf:/home/kafka/config1 -d zcx-kafka-img
```

```
docker exec -it dummy bash
```

```
cp -a /home/kafka/config/* /home/kafka/config1/
```

```
apt-get update
```

```
apt install -y vim
```

- a. Edit the /home/kafka/config1/server.properties file by running the vi command:

Note: The IP address of our zCX host is 129.40.23.76. Make sure you use your IP address.

- i. Update the following values:

```
broker.id=1
zookeeper.connect=129.40.23.76:12181,129.40.23.76:22181,129.40.23.76:3218
1
```

- ii. Add the following lines (under Socket Server Settings):

```
listeners=PLAINTEXT://0.0.0.0:19092
advertised.listeners=PLAINTEXT://129.40.23.76:19092
```

After the updates, the /home/kafka/config1/server.properties file should look like those shown in Example 2-16.

Example 2-16 /home/kafka/config1/server.properties

```
##### Server Basics #####
```

```
# The id of the broker. This must be set to a unique integer for each
broker.
broker.id=1

##### Socket Server Settings #####

# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#   listeners = listener_name://host_name:port
#   EXAMPLE:
#   listeners = PLAINTEXT://your.host.name:9092
#listeners=PLAINTEXT://:9092
listeners=PLAINTEXT://0.0.0.0:19092
advertised.listeners=PLAINTEXT://129.40.23.76:19092

*** OUTPUT OMITTED ***

##### ZooKeeper #####

# ZooKeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=129.40.23.76:12181,129.40.23.76:22181,129.40.23.76:32181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=18000

*** OUTPUT OMITTED ***
```

iii. Run the **exit** command to leave the dummy container.

iv. Run the **docker rm dummy --force** command to delete the dummy container.

2. For kafka-app-2, run the commands that are shown in Example 2-17.

Example 2-17 Create the configuration file for kafka-app-2

```
docker run --name dummy -it --entrypoint /bin/bash -v
kafka2-conf:/home/kafka/config2 -d zcx-kafka-img
```

```
docker exec -it dummy bash
```

```
cp -a /home/kafka/config/* /home/kafka/config2/
```

```
apt-get update
```

```
apt install -y vim
```

a. Edit the `/home/kafka/config2/server.properties` file by running the **vi** command:

Note: The IP address of our zCX host is 129.40.23.76. Make sure you use your IP address.

i. Update the following values:

```
broker.id=2
```

```
zookeeper.connect=129.40.23.76:12181,129.40.23.76:22181,129.40.23.76:32181
```

- ii. Add the following lines (under Socket Server Settings)

```
listeners=PLAINTEXT://0.0.0.0:29092
advertised.listeners=PLAINTEXT://129.40.23.76:29092
```

After the updates, the /home/kafka/config2/server.properties file should look like those lines that are shown in Example 2-18.

Example 2-18 /home/kafka/config/server.properties

```
##### Server Basics #####

# The id of the broker. This must be set to a unique integer for each
broker.
broker.id=2

##### Socket Server Settings #####

# The address the socket server listens on. It will get the value returned
from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#   listeners = listener_name://host_name:port
#   EXAMPLE:
#   listeners = PLAINTEXT://your.host.name:9092
#listeners=PLAINTEXT://:9092
listeners=PLAINTEXT://0.0.0.0:29092
advertised.listeners=PLAINTEXT://129.40.23.76:29092

*** OUTPUT OMITTED ***

##### ZooKeeper #####

# ZooKeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=129.40.23.76:12181,129.40.23.76:22181,129.40.23.76:32181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=18000
```

*** OUTPUT OMITTED ***

-
- iii. Run the **exit** command to leave the dummy container.
 - iv. Run the **docker rm dummy --force** command to delete the dummy container.
3. For kafka-app-3, run the commands that are shown in Example 2-19.

Example 2-19 Create the configuration file for kafka-app-3

```
docker run --name dummy -it --entrypoint /bin/bash -v
kafka3-conf:/home/kafka/config3 -d zcx-kafka-img
```

```
docker exec -it dummy bash
```

```
cp -a /home/kafka/config/* /home/kafka/config3/
```

```
apt-get update
```

```
apt install -y vim
```

-
- a. Edit the `/home/kafka/config3/server.properties` file by running the `vi` command:

Note: The IP address of our zCX host is 129.40.23.76. Make sure you use your IP address.

- i. Update the following values:

```
broker.id=3
zookeeper.connect=129.40.23.76:12181,129.40.23.76:22181,129.40.23.76:32181
```

- ii. Add the following lines (under Socket Server Settings)

```
listeners=PLAINTEXT://0.0.0.0:39092
advertised.listeners=PLAINTEXT://129.40.23.76:39092
```

After the updates, the `/home/kafka/config3/server.properties` file should look like that shown in Example 2-20.

Example 2-20 /home/kafka/config3/server.properties

```
##### Server Basics #####

# The id of the broker. This must be set to a unique integer for each broker.
broker.id=3

##### Socket Server Settings #####

# The address the socket server listens on. It will get the value returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#   listeners = listener_name://host_name:port
#   EXAMPLE:
#   listeners = PLAINTEXT://your.host.name:9092
listeners=PLAINTEXT://:9092
listeners=PLAINTEXT://0.0.0.0:39092
advertised.listeners=PLAINTEXT://129.40.23.76:39092

*** OUTPUT OMITTED ***

##### ZooKeeper #####

# ZooKeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. for example "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=129.40.23.76:12181,129.40.23.76:22181,129.40.23.76:32181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=18000
```

*** OUTPUT OMITTED ***

iii. Run the **exit** command to leave the dummy container.

iv. Run the **docker rm dummy --force** command to delete the dummy container

4. Start your kafka instances up by running the following commands:

```
docker run --name kafka-app-1 -p 19092:19092 -v kafka1-conf:/home/kafka/config
-v kafka1-logs:/home/kafka/logs -d zcx-kafka-img
docker run --name kafka-app-2 -p 29092:29092 -v kafka2-conf:/home/kafka/config
-v kafka2-logs:/home/kafka/logs -d zcx-kafka-img
docker run --name kafka-app-3 -p 39092:39092 -v kafka3-conf:/home/kafka/config
-v kafka3-logs:/home/kafka/logs -d zcx-kafka-img
```

Hint: The `Dockerfile.kafka` file was updated to install the **kafkacat** utility that you can use to test and debug Apache Kafka.

5. Check the Kafka container logs to see whether Kafka was successfully started by running the following command:

```
docker logs kafka-app-1
```

You should receive output that shows brokerID=1 was started, as shown in Example 2-21.

Example 2-21 Logs for kafka-app-1

```
[2020-06-10 11:56:17,413] INFO [TransactionCoordinator id=1] Startup complete.
(kafka.coordinator.transaction.TransactionCoordinator)
[2020-06-10 11:56:17,434] INFO [Transaction Marker Channel Manager 1]: Starting
(kafka.coordinator.transaction.TransactionMarkerChannelManager)
[2020-06-10 11:56:17,476] INFO [ExpirationReaper-1-AlterAcls]: Starting
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2020-06-10 11:56:17,589] INFO [/config/changes-event-process-thread]: Starting
(kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
[2020-06-10 11:56:17,689] INFO [SocketServer brokerId=1] Started data-plane
processors for 1 acceptors (kafka.network.SocketServer)
[2020-06-10 11:56:17,744] INFO Kafka version: 2.5.0
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:17,746] INFO Kafka commitId: 66563e712b0b9f84
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:17,746] INFO Kafka startTimeMs: 1591790177736
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:17,746] INFO [KafkaServer id=1] started
(kafka.server.KafkaServer)
```

6. Check the second Kafka container by running the following command:

```
docker logs kafka-app-2
```

You should receive output that shows brokerID=2 was started, as shown in Example 2-22.

Example 2-22 Logs for kafka-app-2

```
[2020-06-10 11:56:17,982] INFO [TransactionCoordinator id=2] Startup complete.
(kafka.coordinator.transaction.TransactionCoordinator)
[2020-06-10 11:56:18,003] INFO [Transaction Marker Channel Manager 2]: Starting
(kafka.coordinator.transaction.TransactionMarkerChannelManager)
[2020-06-10 11:56:18,032] INFO [ExpirationReaper-2-AlterAcls]: Starting
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2020-06-10 11:56:18,039] INFO [/config/changes-event-process-thread]: Starting
(kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
```

```
[2020-06-10 11:56:18,083] INFO [SocketServer brokerId=2] Started data-plane
processors for 1 acceptors (kafka.network.SocketServer)
[2020-06-10 11:56:18,087] INFO Kafka version: 2.5.0
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:18,090] INFO Kafka commitId: 66563e712b0b9f84
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:18,091] INFO Kafka startTimeMs: 1591790178083
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:18,101] INFO [KafkaServer id=2] started
(kafka.server.KafkaServer)
```

7. Check the third Kafka container by running the following command:

```
docker logs kafka-app-3
```

You should receive output that shows brokerID=3 was started, as shown in Example 2-23.

Example 2-23 Logs for kafka-app-3

```
[2020-06-10 11:56:18,238] INFO [TransactionCoordinator id=3] Startup complete.
(kafka.coordinator.transaction.TransactionCoordinator)
[2020-06-10 11:56:18,242] INFO [Transaction Marker Channel Manager 3]: Starting
(kafka.coordinator.transaction.TransactionMarkerChannelManager)
[2020-06-10 11:56:18,262] INFO [ExpirationReaper-3-AlterAcls]: Starting
(kafka.server.DelayedOperationPurgatory$ExpiredOperationReaper)
[2020-06-10 11:56:18,270] INFO [/config/changes-event-process-thread]: Starting
(kafka.common.ZkNodeChangeNotificationListener$ChangeEventProcessThread)
[2020-06-10 11:56:18,277] INFO [SocketServer brokerId=3] Started data-plane
processors for 1 acceptors (kafka.network.SocketServer)
[2020-06-10 11:56:18,285] INFO Kafka version: 2.5.0
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:18,285] INFO Kafka commitId: 66563e712b0b9f84
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:18,285] INFO Kafka startTimeMs: 1591790178279
(org.apache.kafka.common.utils.AppInfoParser)
[2020-06-10 11:56:18,286] INFO [KafkaServer id=3] started
(kafka.server.KafkaServer)
```

All Kafka container instances are up and running with the correct broker IDs.

Complete the following steps to optionally perform a high availability test to confirm that the instance is up, running, and accessible:

1. Check whether the cluster is healthy. Check the first Kafka instance by running the following command:

```
docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.76:19092 -L"
```

You should receive the output that is shown in Example 2-24.

Example 2-24 Kafka-app-1 check

```
admin@sc74cn09:~$ docker exec -it kafka-app-1 bash -c "kafkacat -b
129.40.23.76:19092 -L"
Metadata for all topics (from broker -1: 129.40.23.76:19092/bootstrap):
3 brokers:
  broker 2 at 129.40.23.76:29092
  broker 3 at 129.40.23.76:39092
  broker 1 at 129.40.23.76:19092
```

0 topics:

2. Check whether the second Kafka instance is accessible by running the following command:

```
docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.76:29092 -L"
```

You should receive the output that is shown in Example 2-25.

Example 2-25 Kafka-app-2 check

```
admin@sc74cn09:~$ docker exec -it kafka-app-2 bash -c "kafkacat -b
129.40.23.76:29092 -L"
Metadata for all topics (from broker -1: 129.40.23.76:29092/bootstrap):
 3 brokers:
    broker 2 at 129.40.23.76:29092
    broker 3 at 129.40.23.76:39092
    broker 1 at 129.40.23.76:19092
 0 topics:
```

3. Check whether the third Kafka instance is accessible by running the following command:

```
docker exec -it kafka-app-3 bash -c "kafkacat -b 129.40.23.76:39092 -L"
```

You should receive the output that is shown in Example 2-26.

Example 2-26 Kafka-app-3 check

```
admin@sc74cn09:~$ docker exec -it kafka-app-3 bash -c "kafkacat -b
129.40.23.76:39092 -L"
Metadata for all topics (from broker -1: 129.40.23.76:39092/bootstrap):
 3 brokers:
    broker 2 at 129.40.23.76:29092
    broker 3 at 129.40.23.76:39092
    broker 1 at 129.40.23.76:19092
 0 topics:
admin@sc74cn09:~$
```

All of the output from each check shows that all three of the Kafka instances can be accessed by way of the zCX IP address.

To verify that the cluster can detect the state of the Kafka instances, take one of the instances down. To take down kafka-app-2, run the following command:

```
docker stop kafka-app-2
```

To check whether the cluster detected a problem on kafka-app-2 and removed this broker from the cluster, run the **kafkacat** command on kafka-app-1 or kafka-app-3 to check the cluster. Run the following command:

```
docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.76:19092 -L"
```

You should receive the output that is shown in Example 2-27.

Example 2-27 Querying Kafka cluster

```
admin@sc74cn09:~$ docker exec -it kafka-app-1 bash -c "kafkacat -b
129.40.23.76:19092 -L"
Metadata for all topics (from broker -1: 129.40.23.76:19092/bootstrap):
 2 brokers:
    broker 3 at 129.40.23.76:39092
```



```
broker 1 at 129.40.23.76:19092
0 topics:
```

To start kafka-app-2 again, run the following command:

```
docker start kafka-app-2
```

Verify whether kafka-app-2 rejoined the cluster by running the following command:

```
docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.76:19092 -L"
```

You should receive the output that is shown in Example 2-28.

Example 2-28 Querying the Kafka cluster

```
admin@sc74cn09:~$ docker exec -it kafka-app-1 bash -c "kafkacat -b
129.40.23.76:19092 -L"
Metadata for all topics (from broker -1: 129.40.23.76:19092/bootstrap):
3 brokers:
  broker 2 at 129.40.23.76:29092
  broker 3 at 129.40.23.76:39092
  broker 1 at 129.40.23.77:19092
0 topics:
```

If you received output that is similar to that shown in Example 2-28, the Kafka cluster is fully operational.

2.6 Kafka configuration on three zCX instances

This section describes how to build a Kafka cluster to achieve high availability. The process is similar to that of deploying the containers in one zCX instance.

Three zCX instances host another Kafka cluster having three ZooKeeper containers and three Kafka containers. This configuration is a high availability solution.

Our example in this section uses the following container names:

- ▶ ZooKeeper:
 - zk-app-1
 - zk-app-2
 - zk-app-3
- ▶ Kafka:
 - kafka-app-1
 - kafka-app-2
 - kafka-app-3

We appended the container instance number to help to identify where each instance are running, as shown in Figure 2-3.

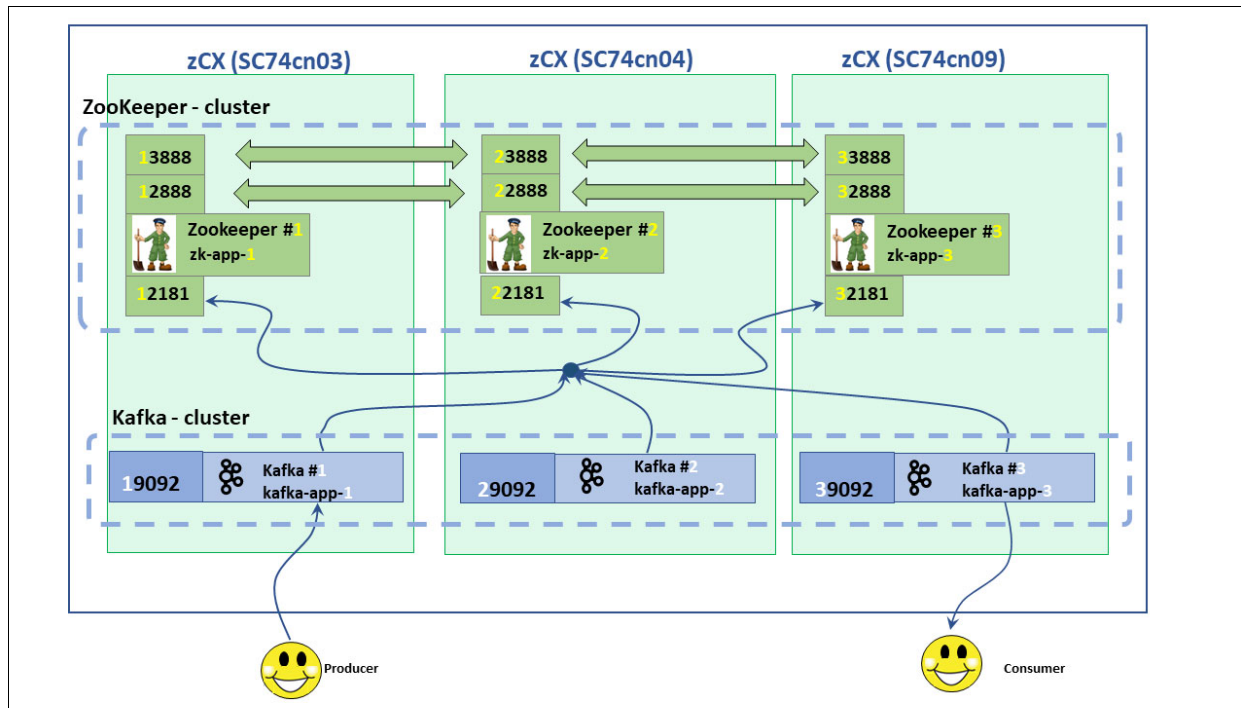


Figure 2-3 Kafka cluster for high availability

Table 2-5 lists our Kafka cluster information. Note the column that is called Instance Location (zCX). This corresponds the zCX instances that are shown in Figure 2-3.

Table 2-5 Kafka cluster information

Container name	Instance number	Instance location (zCX)	Application	Persistent volumes (PV)	Application ports
zk-app-1	1	sc74cn03 (129.40.23.70)	ZooKeeper	zoo1-conf:/conf zoo1-data:/data	12181, 12888, 13888
zk-app-2	2	sc74cn04 (129.40.23.71)	ZooKeeper	zoo2-conf:/conf zoo2-data:/data	22181, 22888, 23888
zk-app-3	3	sc74cn09 (129.40.23.76)	ZooKeeper	zoo3-conf:/conf zoo3-data:/data	32181, 32888, 33888
kafka-app-1	1	sc74cn03 (129.40.23.70)	Kafka	kafka1-conf:/home/kafka/con fig kafka1-logs:/home/kafka/logs	19092
kafka-app-2	2	sc74cn04 (129.40.23.71)	Kafka	kafka2-conf:/home/kafka/con fig kafka2-logs:/home/kafka/logs	29092
kafka-app-3	3	sc74cn09 (129.40.23.76)	Kafka	kafka3-conf:/home/kafka/con fig kafka3-logs:/home/kafka/logs	39092

To easily identify on which zCX host a container will be running, the container names, persistent volumes, and application ports all have the instance number appended to them.

In our example:

- ▶ Containers with instance name #1 will be deployed on sc74cn03 zCX server
- ▶ Containers with instance name #2 will be deployed on sc74cn04 zCX server
- ▶ Containers with instance name #3 will be deployed on sc74cn09 zCX server

Confirming that containers do not exist

If you followed the steps that are described in 2.5.1, “Creating the Kafka cluster” on page 18 to define a Kafka cluster, run the following commands to remove the cluster before a new cluster is defined:

```
docker rm --force zk-app-1
docker rm --force zk-app-2
docker rm --force zk-app-3
docker rm --force kafka-app-1
docker rm --force kafka-app-2
docker rm --force kafka-app-3
```

Creating a ZooKeeper file

No official ZooKeeper Docker image is available on the Docker hub at the time of this writing. The steps that are described in this section build a ZooKeeper image to run on the IBM Z platform. For more information about the latest Dockerfile examples for ZooKeeper, see this [web page](#).

Complete the following steps:

1. Run the commands that are shown in Example 2-29 on your zCX instance to create a folder and download the two required files from GitHub.

Example 2-29 Create the ZooKeeper file

```
mkdir -p /home/admin/zookeeper

cd zookeeper

curl -k -o Dockerfile.zookeeper
https://raw.githubusercontent.com/31z4/zookeeper-docker/master/3.6.1/Dockerfile

curl -k -o docker-entrypoint.sh
https://raw.githubusercontent.com/31z4/zookeeper-docker/master/3.6.1/docker-entrypoint.sh

chmod 755 docker-entrypoint.sh
```

Note: If you encounter problems when the `curl` commands are used to download the required files, download these files from a Linux server and then upload to your zCX instance by running the `scp` or `ssh` commands.

Complete the following steps:

- a. Download the Dockerfile into `/tmp/` and name it `Dockerfile.zookeeper`.
- b. Download `docker-entrypoint.sh` into `/tmp/`.
- c. Run the following commands to upload

```
cat /tmp/Dockerfile.zookeeper | ssh -p 8022 admin@<zcx IP address> "cat > /home/admin/Dockerfile.zookeeper"
```

```
cat /tmp/docker-entrypoint.sh | ssh -p 8022 admin@<zcx IP address> "cat > /home/admin/Docker-entrypoint.sh"
```

The `Dockerfile.zookeeper` file is written to `/home/admin/Dockerfile.zookeeper`.

The `Docker-entrypoint.sh` file is written to `/home/admin/Docker-entrypoint.sh`.

2. Edit the `Dockerfile.zookeeper` file with your details, as shown Example 2-30 and highlighted in **bold**.

Example 2-30 Dockerfile.zookeeper file

FROM s390x/ubuntu:18.04

ENV JAVA_HOME=/usr/lib/jvm/java-11-openjdk-s390x

```
ENV ZOO_CONF_DIR=/conf \  
    ZOO_DATA_DIR=/data \  
    ZOO_DATA_LOG_DIR=/data/log \  
    ZOO_LOG_DIR=/logs \  
    ZOO_TICK_TIME=2000 \  
    ZOO_INIT_LIMIT=5 \  
    ZOO_SYNC_LIMIT=2 \  
    ZOO_AUTOPURGE_PURGEINTERVAL=0 \  
    ZOO_AUTOPURGE_SNAPRETAINCOUNT=3 \  
    ZOO_MAX_CLIENT_CNXNS=60 \  
    ZOO_STANDALONE_ENABLED=true \  
    ZOO_ADMIN_SERVER_ENABLED=true
```

```
# Add a user with an explicit UID/GID and create necessary directories
```

```
RUN set -eux; \  
    groupadd -r zookeeper --gid=1000; \  
    useradd -r -g zookeeper --uid=1000 zookeeper; \  
    mkdir -p "$ZOO_DATA_LOG_DIR" "$ZOO_DATA_DIR" "$ZOO_CONF_DIR" "$ZOO_LOG_DIR"; \  
    chown zookeeper:zookeeper "$ZOO_DATA_LOG_DIR" "$ZOO_DATA_DIR" "$ZOO_CONF_DIR" \  
    "$ZOO_LOG_DIR"
```

```
# Install required packages
```

```
RUN set -eux; \  
    apt-get update; \  
    DEBIAN_FRONTEND=noninteractive \  
    apt-get install -y --no-install-recommends \  
    openjdk-11-jre \  
    &
```

```

tar \
wget \
vim \
ca-certificates \
dirmngr \
gosu \
gnupg \
netcat \
rm -rf /var/lib/apt/lists/*; \
# Verify that gosu binary works
gosu nobody true

ARG GPG_KEY=BBE7232D7991050B54C8EA0ADC08637CA615D22C
ARG SHORT_DISTRO_NAME=zookeeper-3.6.1
ARG DISTRO_NAME=apache-zookeeper-3.6.1-bin

# Download Apache ZooKeeper, verify its PGP signature, untar and clean up
RUN set -eux; \
ddist() { \
    local f="$1"; shift; \
    local distFile="$1"; shift; \
    local success=; \
    local distUrl=; \
    for distUrl in \
        'https://www.apache.org/dyn/closer.cgi?action=download&filename=' \
        https://www-us.apache.org/dist/ \
        https://www.apache.org/dist/ \
        https://archive.apache.org/dist/ \
    ; do \
        if wget -q -O "$f" "$distUrl$distFile" && [ -s "$f" ]; then \
            success=1; \
            break; \
        fi; \
    done; \
    [ -n "$success" ]; \
}; \
ddist "$DISTRO_NAME.tar.gz"
"zookeeper/$SHORT_DISTRO_NAME/$DISTRO_NAME.tar.gz"; \
ddist "$DISTRO_NAME.tar.gz.asc"
"zookeeper/$SHORT_DISTRO_NAME/$DISTRO_NAME.tar.gz.asc"; \
export GNUPGHOME="$(mktemp -d)"; \
gpg --keyserver ha.pool.sks-keyservers.net --recv-key "$GPG_KEY" || \
gpg --keyserver pgp.mit.edu --recv-keys "$GPG_KEY" || \
gpg --keyserver keyserver.pgp.com --recv-keys "$GPG_KEY"; \
gpg --batch --verify "$DISTRO_NAME.tar.gz.asc" "$DISTRO_NAME.tar.gz"; \
tar -zxf "$DISTRO_NAME.tar.gz"; \
mv "$DISTRO_NAME/conf/*" "$ZOO_CONF_DIR"; \
rm -rf "$GNUPGHOME" "$DISTRO_NAME.tar.gz" "$DISTRO_NAME.tar.gz.asc"; \
chown -R zookeeper:zookeeper "/$DISTRO_NAME"

WORKDIR $DISTRO_NAME
VOLUME ["$ZOO_DATA_DIR", "$ZOO_DATA_LOG_DIR", "$ZOO_LOG_DIR"]

#EXPOSE 2181 2888 3888 8080

```

```
ENV PATH=$PATH:/$DISTRO_NAME/bin \  
    ZOOCFGDIR=$ZOO_CONF_DIR  
  
COPY docker-entrypoint.sh /  
ENTRYPOINT ["/docker-entrypoint.sh"]  
CMD ["zkServer.sh", "start-foreground"]
```

Note: Writing EXPOSE in the Dockerfile is merely a hint that a specific port is useful. Docker does not use that information.

3. Run the following command to build the ZooKeeper image:

```
docker build -t zcx-zookeeper-img -f Dockerfile.zookeeper
```

We named the new image zcx-zookeeper-img.

4. Create two persistent volumes for each ZooKeeper instance by running the following commands:

- For zk-app-1, run the following commands that are connected to sc74cn03 server:

```
docker volume create zoo1-conf  
docker volume create zoo1-data
```

- For zk-app-2, run the following commands that are connected to sc74cn04 server:

```
docker volume create zoo2-conf  
docker volume create zoo2-data
```

- For zk-app-3, run the following commands that are connected to sc74cn09 server:

```
docker volume create zoo3-conf  
docker volume create zoo3-data
```

Complete the following steps to populate the volumes for each container instance. We use a dummy temporary image in our example:

1. For zk-app-1, run the commands that are shown in Example 2-31 on the sc74cn03 server.

Example 2-31 Populate the volumes for zk-app-1

```
docker run --name dummy -v zoo1-conf:/conf1 -v zoo1-data:/data1 -d  
zcx-zookeeper-img
```

```
docker exec -it dummy bash
```

```
cp -a /conf/* /conf1/
```

```
cp -a /data/* /data1/
```

```
apt-get update
```

```
apt install -y vim
```

Edit the /conf1/zoo.cfg file by running the vi command:

- a. Delete the following line:

```
server.1=localhost:2888:3888;2181 line
```

- b. Update the following values:

```
initLimit=10
syncLimit=5
standaloneEnabled=false
```

- c. Add the following lines:

```
server.1=0.0.0.0:12888:13888;12181
server.2=129.40.23.71:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181
```

After the updates, the `zoo.cfg` file should look similar to that shown in Example 2-32.

Example 2-32 /conf1/zoo.cfg file

```
dataDir=/data
dataLogDir=/dataLog
tickTime=2000
initLimit=10
syncLimit=5
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
standaloneEnabled=false
admin.enableServer=true
server.1=0.0.0.0:12888:13888;12181
server.2=129.40.23.71:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181
```

- d. Edit the `/data1/myid` file by running the `vi` command and update with the number 1 for our first ZooKeeper instance.
- e. Run the `exit` command to leave the dummy container.
- f. Run the `docker rm dummy --force` command to delete the dummy container.
2. For `zk-app-2`, run the commands that are shown in Example 2-33 on the `sc74cn04` server.

Example 2-33 Populate the volumes for zk-app-2

```
docker run --name dummy -v zoo2-conf:/conf2 -v zoo2-data:/data2 -d
zcx-zookeeper-img
```

```
docker exec -it dummy bash
```

```
cp -a /conf/* /conf2/
```

```
cp -a /data/* /data2/
```

```
apt-get update
```

```
apt install -y vim
```

Edit the `/conf2/zoo.cfg` file by running the `vi` command:

- a. Delete the following line:

```
server.1=localhost:2888:3888;2181 line
```

- b. Update the following values:

```
initLimit=10
```

```
syncLimit=5
standaloneEnabled=false
```

- c. Add the following lines:

```
server.1=129.40.23.70:12888:13888;12181
server.2=0.0.0.0:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181
```

After the updates, the `zoo.cfg` file should look similar to that shown in Example 2-34.

Example 2-34 /conf2/zoo.cfg file

```
dataDir=/data
dataLogDir=/data/log
tickTime=2000
initLimit=10
syncLimit=5
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
standaloneEnabled=false
admin.enableServer=true
server.1=129.40.23.70:12888:13888;12181
server.2=0.0.0.0:22888:23888;22181
server.3=129.40.23.76:32888:33888;32181
```

- d. Edit the `/data2/myid` file by running the `vi` command and update with the number 2 for our second ZooKeeper instance.
- e. Run the `exit` command to leave the dummy container.
- f. Run the `docker rm dummy --force` command to delete the dummy container.
3. For `zk-app-3`, run the commands that are shown in Example 2-35 on the `sc74cn09` server.

Example 2-35 Populate the volumes for zk-app-3

```
docker run --name dummy -v zoo3-conf:/conf3 -v zoo3-data:/data3 -d
zcx-zookeeper-img
```

```
docker exec -it dummy bash
```

```
cp -a /conf/* /conf3/
```

```
cp -a /data/* /data3/
```

```
apt-get update
```

```
apt install -y vim
```

- a. Edit the `/conf3/zoo.cfg` file by running the `vi` command:
- Delete the following line:

```
server.1=localhost:2888:3888;2181
```

 line
 - Update the following values:

```
initLimit=10
syncLimit=5
standaloneEnabled=false
```


iii. Add the following lines:

```
server.1=129.40.23.70:12888:13888;12181
server.2=129.40.23.71:22888:23888;22181
server.3=0.0.0.0:32888:33888;32181
```

After the updates, the `zoo.cfg` file should look similar to that shown in Example 2-8.

Example 2-36 /conf3/zoo.cfg file

```
dataDir=/data
dataLogDir=/dataLog
tickTime=2000
initLimit=10
syncLimit=5
autopurge.snapRetainCount=3
autopurge.purgeInterval=0
maxClientCnxns=60
standaloneEnabled=false
admin.enableServer=true
server.1=129.40.23.70:12888:13888;12181
server.2=129.40.23.71:22888:23888;22181
server.3=0.0.0.0:32888:33888;32181
```

- b. Edit the `/data3/myid` file by running the `vi` command and update with the number 3 for our third ZooKeeper instance.
 - c. Run the `exit` command to leave the dummy container.
 - d. Run the `docker rm dummy --force` command to delete the dummy container.
4. Start your three ZooKeeper instances. Because ZooKeeper ‘fails fast’, it is better to always restart it. We appended `--restart always` in the Docker run commands, as shown in Example 2-37.

Example 2-37 Docker run commands

```
# On sc74cn03 server:
docker run --name zk-app-1 -p 12181:12181 -p 12888:12888 -p 13888:13888 -v
zoo1-conf:/conf:rw -v zoo1-data:/data:rw --restart always -d zcx-zookeeper-img

# On sc74cn04 server:
docker run --name zk-app-2 -p 22181:22181 -p 22888:22888 -p 23888:23888 -v
zoo2-conf:/conf:rw -v zoo2-data:/data:rw --restart always -d zcx-zookeeper-img

# On sc74cn09 server:
docker run --name zk-app-3 -p 32181:32181 -p 32888:32888 -p 33888:33888 -v
zoo3-conf:/conf:rw -v zoo3-data:/data:rw --restart always -d zcx-zookeeper-img
```

5. Run the following commands to verify the start of each container:
- For the `sc74cn03` server:
`docker logs zk-app-1`
 - For the `sc74cn04` server:
`docker logs zk-app-2`
 - For the `sc74cn09` server:
`docker logs zk-app-3`

For each command, verify that you receive a message as shown in Example 2-38.

Example 2-38 Example messages of successful start of each container

```
*** OUTPUT OMITTED ***
2020-06-09 16:41:30,773 [myid:1] - INFO [main:ContextHandler@825] - Started
o.e.j.s.ServletContextHandler@33ecda92{/,null,AVAILABLE}
2020-06-09 16:41:30,778 [myid:1] - INFO [main:AbstractConnector@330] - Started
ServerConnector@4c60d6e9{HTTP/1.1,[http/1.1]}{0.0.0.0:8080}
2020-06-09 16:41:30,778 [myid:1] - INFO [main:Server@399] - Started @975ms
2020-06-09 16:41:30,778 [myid:1] - INFO [main:JettyAdminServer@178] - Started
AdminServer on address 0.0.0.0, port 8080 and command URL /commands
*** OUTPUT OMITTED ***
```

6. For zk-app-1 instance on the sc74cn03 server, run the following commands to confirm ZooKeeper is operational:

```
docker exec -it zk-app-1 bash -c "wget -O -
http://localhost:8080/commands/stats"
```

You should receive output similar to that shown in Example 2-39. Highlighted in **bold** is the state of each instance. One instance should be a leader and the others followers.

Example 2-39 Confirmation of follower instance

```
admin@sc74cn03:~/zookeeper$ docker exec -it zk-app-1 bash -c "wget -O -
http://localhost:8080/commands/stats"
--2020-08-18 18:04:55-- http://localhost:8080/commands/stats
Resolving localhost (localhost)... 127.0.0.1, ::1
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 904 [application/json]
Saving to: 'STDOUT'
```

```
-
0%[
] 0 --.-KB/s {
"version" : "3.6.1--104dcb3e3fb464b30c5186d229e00af9f332524b, built on
04/21/2020 15:01 GMT",
"read_only" : false,
"server_stats" : {
  "packets_sent" : 0,
  "packets_received" : 0,
  "fsync_threshold_exceed_count" : 0,
  "client_response_stats" : {
    "last_buffer_size" : -1,
    "min_buffer_size" : -1,
    "max_buffer_size" : -1
  },
  "uptime" : 428981,
  "provider_null" : false,
  "server_state" : "follower",
  "outstanding_requests" : 0,
  "min_latency" : 0,
  "avg_latency" : 0.0,
  "max_latency" : 0,
  "data_dir_size" : 0,
  "log_dir_size" : 459,
```

```

        "last_processed_zxid" : 0,
        "num_alive_client_connections" : 0
    },
    "client_response" : {
        "last_buffer_size" : -1,
        "min_buffer_size" : -1,
        "max_buffer_size" : -1
    },
    "node_count" : 5,
    "connections" : [ ],
    "secure_connections" : [ ],
    "command" : "stats",
    "error" : null
}
-
100%[=====>]    904 --.-KB/s
in 0s

```

2020-08-18 18:04:56 (484 MB/s) - written to stdout [904/904]

7. For zk-app-2 instance on the sc74cn04 server, run the following commands to confirm ZooKeeper is operational:

```

docker exec -it zk-app-2 bash -c "wget -O -
http://localhost:8080/commands/stats"

```

You should receive similar output as shown in Example 2-40. Highlighted in **red** is the state of each instance. One instance should be a leader and the others followers.

Example 2-40 Confirmation of leader instance

```

admin@sc74cn04:~/zookeeper$ docker exec -it zk-app-2 bash -c "wget -O -
http://localhost:8080/commands/stats"
--2020-08-18 18:06:28-- http://localhost:8080/commands/stats
Resolving localhost (localhost)... 127.0.0.1, ::1
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 1023 [application/json]
Saving to: 'STDOUT'

```

```

-
0%[
]    0 --.-KB/s    {
  "version" : "3.6.1--104dcb3e3fb464b30c5186d229e00af9f332524b, built on
04/21/2020 15:01 GMT",
  "read_only" : false,
  "server_stats" : {
    "packets_sent" : 0,
    "packets_received" : 0,
    "fsync_threshold_exceed_count" : 0,
    "client_response_stats" : {
      "last_buffer_size" : -1,
      "min_buffer_size" : -1,
      "max_buffer_size" : -1
    },
    "uptime" : 521767,
    "provider_null" : false,
    "server_state" : "leader",
    "outstanding_requests" : 0,

```

```

    "min_latency" : 0,
    "avg_latency" : 0.0,
    "max_latency" : 0,
    "data_dir_size" : 0,
    "log_dir_size" : 459,
    "last_processed_zxid" : 4294967296,
    "num_alive_client_connections" : 0
  },
  "client_response" : {
    "last_buffer_size" : -1,
    "min_buffer_size" : -1,
    "max_buffer_size" : -1
  },
  "proposal_stats" : {
    "last_buffer_size" : -1,
    "min_buffer_size" : -1,
    "max_buffer_size" : -1
  },
  "node_count" : 5,
  "connections" : [ ],
  "secure_connections" : [ ],
  "command" : "stats",
  "error" : null
}
-
100%[=====>] 1023 --.-KB/s
in 0s

```

2020-08-18 18:06:28 (481 MB/s) - written to stdout [1023/1023]

8. For the zk-app-3 instance on the sc74cn09 server, run the following commands to confirm ZooKeeper is operational:

```

docker exec -it zk-app-3 bash -c "wget -O -
http://localhost:8080/commands/stats"

```

You should receive output shown in Example 2-41. Highlighted in **red** is the state of each instance. One instance should be a leader and the others followers.

Example 2-41 Confirmation of second follower instance

```

admin@sc74cn09:~/zookeeper$ docker exec -it zk-app-3 bash -c "wget -O -
http://localhost:8080/commands/stats"
--2020-08-18 18:10:27-- http://localhost:8080/commands/stats
Resolving localhost (localhost)... 127.0.0.1, ::1
Connecting to localhost (localhost)|127.0.0.1|:8080... connected.
HTTP request sent, awaiting response... 200 OK
Length: 914 [application/json]
Saving to: 'STDOUT'

```

```

-
0%[
] 0 --.-KB/s {
  "version" : "3.6.1--104dcb3e3fb464b30c5186d229e00af9f332524b, built on
04/21/2020 15:01 GMT",
  "read_only" : false,
  "server_stats" : {
    "packets_sent" : 0,
    "packets_received" : 0,

```

```

    "fsync_threshold_exceed_count" : 0,
    "client_response_stats" : {
      "last_buffer_size" : -1,
      "min_buffer_size" : -1,
      "max_buffer_size" : -1
    },
    "provider_null" : false,
    "uptime" : 603836,
    "server_state" : "follower",
    "outstanding_requests" : 0,
    "min_latency" : 0,
    "avg_latency" : 0.0,
    "max_latency" : 0,
    "data_dir_size" : 0,
    "log_dir_size" : 1100,
    "last_processed_zxid" : 4294967296,
    "num_alive_client_connections" : 0
  },
  "client_response" : {
    "last_buffer_size" : -1,
    "min_buffer_size" : -1,
    "max_buffer_size" : -1
  },
  "node_count" : 5,
  "connections" : [ ],
  "secure_connections" : [ ],
  "command" : "stats",
  "error" : null
}
-
100%[=====] 914 --.-KB/s
in 0s

```

2020-08-18 18:10:27 (383 MB/s) - written to stdout [914/914]

The ZooKeeper installation is completed. Next, you start the installation of Kafka in a high availability use case.

Kafka Docker images

In this section, we describe how to create images to install Kafka.

Complete the following steps on sc74cn03, sc74cn04, and sc74cn09:

1. Run the following commands to create a folder for the Kafka image:

```

mkdir /home/admin/kafka
cd /home/admin/kafka

```

2. Download the Kafka Dockerfile by running the following command:

Note: You can check latest Dockerfile example for Kafka in <https://github.com/linux-on-ibm-z/dockerfile-examples>.

```

curl -k -o Dockerfile.kafka
https://raw.githubusercontent.com/linux-on-ibm-z/dockerfile-examples/master/Apa
cheKafka/Dockerfile

```

Note: If you encounter problems when the **curl** commands are run to download the required files, download these files from a Linux server and then upload to your zCX instance by running the **scp** or **ssh** commands.

Complete the following steps:

a. Download the Dockerfile into /tmp/ and name it Dockerfile.kafka.

b. Run the following commands to upload:

```
cat /tmp/Dockerfile.kafka | ssh -p 8022 admin@<zcx IP address> "cat >
/home/admin/Dockerfile.kafka"
```

The Dockerfile.kafka file is written to /home/admin/Dockerfile.kafka.

3. Edit the Dockerfile.kafka file to look similar to that shown in Example 2-42. Highlighted in **bold** is what is required to update, add, or remove.

Example 2-42 Dockerfile.kafka

```
# © Copyright IBM Corporation 2017, 2020.
# LICENSE: Apache License, Version 2.0
(http://www.apache.org/licenses/LICENSE-2.0)

##### dockerfile for Apache Kafka version 2.5.0 #####
#
# This Dockerfile builds a basic installation of Apache Kafka.
#
# Kafka is run as a cluster on one or more servers. The Kafka cluster stores
streams of records in categories called topics.
# Each record consists of a key, a value, and a timestamp.
# In Kafka the communication between the clients and the servers is done with a
simple, high-performance, language agnostic TCP protocol.
#
# To build this image, from the directory containing this Dockerfile
# (assuming that the file is named Dockerfile):
# docker build -t <image_name> .
#
# To Start Apache Kafka run the below command:
# docker run --name <container_name> -d <image>
#
# To check Apache kafka is running, Enter below command:
# docker exec <container_id of kafka> <any kafka related command>
# Eg. To list topic and message files:
# docker exec <container_id of kafka> bin/kafka-topics.sh --list --zookeeper
localhost:2181
#
# Reference:
# http://kafka.apache.org/
# https://kafka.apache.org/quickstart
#
#####
###

# Base Image
```

```

FROM s390x/ubuntu:20.04

# The author
LABEL maintainer="LoZ Open Source Ecosystem
(https://www.ibm.com/developerworks/community/groups/community/lozopensource)"

ENV SOURCE_DIR=/home/
ENV JAVA_HOME=/home/jdk-11.0.5+10
ENV PATH=$PATH:$SOURCE_DIR:$JAVA_HOME/bin
ENV VERSION=2.12-2.5.0

WORKDIR $SOURCE_DIR

# Install dependencies
RUN apt-get update && apt-get -y install \
    git \
    unzip \
    wget \
    vim \
    net-tools \
    kafkacat \
# Download Adopt JDK
    && wget
https://github.com/AdoptOpenJDK/openjdk11-binaries/releases/download/jdk-11.0.5
%2B10/OpenJDK11U-jdk_s390x_linux_hotspot_11.0.5_10.tar.gz \
    && tar -xvzf OpenJDK11U-jdk_s390x_linux_hotspot_11.0.5_10.tar.gz \
    && rm OpenJDK11U-jdk_s390x_linux_hotspot_11.0.5_10.tar.gz \
# Download the Apache Kafka Binary
    && wget http://mirrors.estointernet.in/apache/kafka/2.5.0/kafka_${VERSION}.tgz
\
    && tar -xvzf kafka_${VERSION}.tgz \
    && rm kafka_${VERSION}.tgz \
    && mv kafka_${VERSION} kafka
# Expose ports for Apache ZooKeeper and kafka
#EXPOSE 2181 9092

WORKDIR $SOURCE_DIR/kafka/

# start zookeeper and kafka server
CMD bin/zookeeper-server-start.sh -daemon config/zookeeper.properties && sleep
20 && bin/kafka-server-start.sh config/server.properties > /dev/null

# End of Dockerfile

```

Note: Writing EXPOSE in Dockerfile is a hint that a specific port is useful. Docker does not use that information.

4. To build a new Kafka image and tag it as zcx-kafka-img, run the following command:

```
docker build -t zcx-kafka-img -f Dockerfile.kafka
```

You receive the follow messages if the build completes successfully:

```

Successfully built 3ac1c714ee08
Successfully tagged zcx-kafka-img:latest

```

Note: Remember to build this image on all zCX hosts.

5. Define kafka volume groups for each Kafka container instance by running the following commands:

- On sc74cn03:
 docker volume create kafka1-conf
 docker volume create kafka1-logs
- On sc74cn04:
 docker volume create kafka2-conf
 docker volume create kafka2-logs
- On sc74cn09:
 docker volume create kafka3-conf
 docker volume create kafka3-logs

Running these commands makes the configuration files persistent for Kafka, which allows your system to survive during an upgrade. For more information about persistent volumes, see *Getting started with z/OS Container Extensions and Docker*, SG24-8457.

Complete the following steps to create the configuration file for each Kafka container image:

1. For kafka-app-1, run the commands on sc74cn03 that are shown in Example 2-43.

Example 2-43 Create the configuration file for kafka-app1 on sc74cn03

```
docker run --name dummy -it --entrypoint /bin/bash -v
kafka1-conf:/home/kafka/config1 -v kafka1-logs:/home/kafka/logs1 -d
zcx-kafka-img

docker exec -it dummy bash

cp -a /home/kafka/config/* /home/kafka/config1/

cp -a /home/kafka/logs/* /home/kafka/logs1/

apt-get update

apt install -y vim
```

- a. Edit /home/kafka/config1/server.properties file by running the vi command:

Note: Consider the following points:

- 129.40.23.70 is the IP address of sc74cn03
- 129.40.23.71 is the IP address of sc74cn04
- 129.40.23.76 is the IP address of sc74cn09

- i. Update the following values:

```
broker.id=1
zookeeper.connect=129.40.23.70:12181,129.40.23.71:22181,129.40.23.76:32181
```


- ii. Add the following lines (under Socket Server Settings):

```
listeners=PLAINTEXT://0.0.0.0:19092
advertised.listeners=PLAINTEXT://129.40.23.70:19092
```

After the updates, the `/home/kafka/config1/server.properties` file should look like the example that is shown in Example 2-16.

Example 2-44 /home/kafka/config1/server.properties file

```
##### Server Basics #####
```

```
# The id of the broker. This must be set to a unique integer for each
broker.
broker.id=1
```

```
##### Socket Server Settings #####
```

```
# The address the socket server listens on. It will get the value
returned from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#     listeners = listener_name://host_name:port
#   EXAMPLE:
#     listeners = PLAINTEXT://your.host.name:9092
#listeners=PLAINTEXT://:9092
listeners=PLAINTEXT://0.0.0.0:19092
advertised.listeners=PLAINTEXT://129.40.23.70:19092
```

***** OUTPUT OMITTED *****

```
##### ZooKeeper #####
```

```
# ZooKeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify
the
# root directory for all kafka znodes.
zookeeper.connect=129.40.23.70:12181,129.40.23.71:22181,129.40.23.76:32181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=18000
```

***** OUTPUT OMITTED *****

-
- iii. Issue **exit** command to leave the dummy container
 - iv. Issue **docker rm dummy --force** command to delete the dummy container.
2. For `kafka-app-2`, run the commands shown in Example 2-45 on `sc74cn04`:

Example 2-45 Create the configuration file for `kafka-app-2` on `sc74cn04`

```
docker run --name dummy -it --entrypoint /bin/bash -v
kafka2-conf:/home/kafka/config2 -v kafka2-logs:/home/kafka/logs2 -d
zcx-kafka-img
```

```
docker exec -it dummy bash
```

```
cp -a /home/kafka/config/* /home/kafka/config2/
```

```
cp -a /home/kafka/logs/* /home/kafka/logs2/
```

```
apt-get update
```

```
apt install -y vim
```

a. Edit the `/home/kafka/config2/server.properties` file by running the `vi` command:

Note: Consider the following points:

- ▶ 129.40.23.70 is the IP address of sc74cn03
- ▶ 129.40.23.71 is the IP address of sc74cn04
- ▶ 129.40.23.76 is the IP address of sc74cn09

i. Update the following values:

```
broker.id=2
```

```
zookeeper.connect=129.40.23.70:12181,129.40.23.71:22181,129.40.23.76:32181
```

ii. Add the following lines (under Socket Server Settings):

```
listeners=PLAINTEXT://0.0.0.0:29092
```

```
advertised.listeners=PLAINTEXT://129.40.23.71:29092
```

After the updates, the `/home/kafka/config2/server.properties` file should look like the example that is shown in Example 2-46.

Example 2-46 /home/kafka/config2/server.properties file

```
##### Server Basics #####
```

```
# The id of the broker. This must be set to a unique integer for each broker.
```

```
broker.id=2
```

```
##### Socket Server Settings #####
```

```
# The address the socket server listens on. It will get the value returned from
```

```
# java.net.InetAddress.getCanonicalHostName() if not configured.
```

```
# FORMAT:
```

```
#   listeners = listener_name://host_name:port
```

```
# EXAMPLE:
```

```
#   listeners = PLAINTEXT://your.host.name:9092
```

```
#listeners=PLAINTEXT://:9092
```

```
listeners=PLAINTEXT://0.0.0.0:29092
```

```
advertised.listeners=PLAINTEXT://129.40.23.71:29092
```

***** OUTPUT OMITTED *****

```
##### ZooKeeper #####
```

```
# ZooKeeper connection string (see zookeeper docs for details).
```

```
# This is a comma separated host:port pairs, each corresponding to a zk server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
```

```
# You can also append an optional chroot string to the urls to specify the
```

```
# root directory for all kafka znodes.
zookeeper.connect=129.40.23.70:12181,129.40.23.71:22181,129.40.23.76:32181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=18000
```

*** OUTPUT OMITTED ***

- iii. Run the **exit** command to leave the dummy container.
 - iv. Run the **docker rm dummy --force** command to delete the dummy container.
3. For **kafka-app-3**, run the commands on **sc74cn09** that are shown in Example 2-47.

Example 2-47 Create the configuration file for kafka-app-3 on sc74cn09

```
docker run --name dummy -it --entrypoint /bin/bash -v
kafka3-conf:/home/kafka/config3 -v kafka3-logs:/home/kafka/logs3 -d
zcx-kafka-img
```

```
docker exec -it dummy bash
```

```
cp -a /home/kafka/config/* /home/kafka/config3/
```

```
cp -a /home/kafka/logs/* /home/kafka/logs3/
```

```
apt-get update
```

```
apt install -y vim
```

- a. Edit the **/home/kafka/config3/server.properties** file by running the **vi** command:

Note: Consider the following points:

- ▶ 129.40.23.70 is the IP address of **sc74cn03**
- ▶ 129.40.23.71 is the IP address of **sc74cn04**
- ▶ 129.40.23.76 is the IP address of **sc74cn09**

- i. Update the following values:

```
broker.id=3
zookeeper.connect=129.40.23.70:12181,129.40.23.71:22181,129.40.23.76:32181
```

- ii. Add the following lines (under Socket Server Settings):

```
listeners=PLAINTEXT://0.0.0.0:39092
advertised.listeners=PLAINTEXT://129.40.23.76:39092
```

After the updates, the **/home/kafka/config3/server.properties** file should look like Example 2-48.

Example 2-48 /home/kafka/config3/server.properties

```
##### Server Basics #####
```

```
# The id of the broker. This must be set to a unique integer for each
broker.
broker.id=3
```

```
##### Socket Server Settings #####

# The address the socket server listens on. It will get the value returned
# from
# java.net.InetAddress.getCanonicalHostName() if not configured.
#   FORMAT:
#   listeners = listener_name://host_name:port
#   EXAMPLE:
#   listeners = PLAINTEXT://your.host.name:9092
#listeners=PLAINTEXT://:9092
listeners=PLAINTEXT://0.0.0.0:39092
advertised.listeners=PLAINTEXT://129.40.23.76:39092

*** OUTPUT OMITTED ***

##### ZooKeeper #####

# ZooKeeper connection string (see zookeeper docs for details).
# This is a comma separated host:port pairs, each corresponding to a zk
# server. e.g. "127.0.0.1:3000,127.0.0.1:3001,127.0.0.1:3002".
# You can also append an optional chroot string to the urls to specify the
# root directory for all kafka znodes.
zookeeper.connect=129.40.23.70:12181,129.40.23.71:22181,129.40.23.76:32181

# Timeout in ms for connecting to zookeeper
zookeeper.connection.timeout.ms=18000

*** OUTPUT OMITTED ***
```

iii. Run the **exit** command to leave the dummy container.

iv. Run the **docker rm dummy --force** command to delete the dummy container.

4. Start your kafka instances by running the following commands:

– On sc74cn03:

```
docker run --name kafka-app-1 -p 19092:19092 -v
kafka1-conf:/home/kafka/config -v kafka1-logs:/home/kafka/logs -d
zcx-kafka-img
```

– On sc74cn04:

```
docker run --name kafka-app-2 -p 29092:29092 -v
kafka2-conf:/home/kafka/config -v kafka2-logs:/home/kafka/logs -d
zcx-kafka-img
```

– On sc74cn09:

```
docker run --name kafka-app-3 -p 39092:39092 -v
kafka3-conf:/home/kafka/config -v kafka3-logs:/home/kafka/logs -d
zcx-kafka-img
```

Hint: The Dockerfile.kafka file was updated to install the **kafkacat** utility that you can use to test and debug Apache Kafka.

5. Check the Kafka container logs to see whether kafka was successfully started by running the following command:

- On sc74cn03:
`docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.70:19092 -L"`
- On sc74cn04:
`docker exec -it kafka-app-2 bash -c "kafkacat -b 129.40.23.71:29092 -L"`
- On sc74cn09:
`docker exec -it kafka-app-3 bash -c "kafkacat -b 129.40.23.76:39092 -L"`

The Output should be:

```
Metadata for all topics (from broker 1: 129.40.23.70:19092/1):
3 brokers:
  broker 2 at 129.40.23.71:29092
  broker 3 at 129.40.23.76:39092
  broker 1 at 129.40.23.70:19092 (controller)
0 topics:
```

Complete the following steps to optionally perform a high availability test to ensure that the instance is up and running and accessible:

1. Check whether the cluster is healthy. Check the first Kafka instance by running the following command on sc74cn03:

```
docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.70:19092 -L"
```

You should receive the output that is shown in Example 2-49.

Example 2-49 Kafka-app-1 check

```
Metadata for all topics (from broker 1: 129.40.23.70:19092/1):
3 brokers:
  broker 2 at 129.40.23.71:29092
  broker 3 at 129.40.23.76:39092
  broker 1 at 129.40.23.70:19092 (controller)
0 topics:
```

2. To verify that the cluster can detect the state of the Kafka instances, take one of the instances down. To take down kafka-app-2, run the following command on sc74cn04:

```
docker stop kafka-app-2
```

3. To check whether the cluster detected a problem on kafka-app-2 and removed this broker from the cluster, run the **kafkacat** command on kafka-app-1 to check the cluster.

Run the following command on sc74cn03:

```
docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.70:19092 -L"
```

You should receive the output shown in Example 2-27.

Example 2-50 Querying Kafka cluster

```
admin@sc74cn03:~/kafka$ docker exec -it kafka-app-1 bash -c "kafkacat -b
129.40.23.70:19092 -L"
Metadata for all topics (from broker 1: 129.40.23.70:19092/1):
2 brokers:
  broker 3 at 129.40.23.76:39092
  broker 1 at 129.40.23.70:19092 (controller)
```

0 topics:

4. To start kafka-app-2 again, run the following command on sc74cn04:
`docker start kafka-app-2`
5. Verify whether kafka-app-2 rejoined the cluster by running the following command on sc74cn03:
`docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.70:19092 -L"`
You should receive the output shown in Example 2-51.

Example 2-51 Querying the Kafka cluster

```
admin@sc74cn03:~/kafka$ docker exec -it kafka-app-1 bash -c "kafkacat -b
129.40.23.70:19092 -L"
Metadata for all topics (from broker 1: 129.40.23.70:19092/1):
 3 brokers:
   broker 2 at 129.40.23.71:29092
   broker 3 at 129.40.23.76:39092
   broker 1 at 129.40.23.70:19092 (controller)
 0 topics::
```

If you received the output similar to that shown in Example 2-51, the Kafka cluster is fully operational.

2.7 Diagnostic commands

You might encounter issues that must be diagnosed and corrected. This section describes some basic commands that can be used for ZooKeeper and Kafka that assist you in diagnosing and correcting these issues.

2.7.1 ZooKeeper

These commands must be run inside one of the ZooKeeper instances (in our example, zk-app-1, 2, or 3). Run the `docker exec -it zk-app-1 bash` command to connect to the zk-app-1 container and then run the following commands:

- ▶ To list active brokers:
`/apache-zookeeper-3.6.1-bin/bin/zkCli.sh -server 129.40.23.76:12181 ls /brokers/ids`
- ▶ To get detailed information about a specific broker ID "1" (broker numbers can be found by running the previous command):
`/apache-zookeeper-3.6.1-bin/bin/zkCli.sh -server 129.40.23.76:12181 get /brokers/ids/1`
Repeat each command for each of the brokers (broker ID 2 and broker ID 3).
- ▶ To list topics:
`/apache-zookeeper-3.6.1-bin/bin/zkCli.sh -server 129.40.23.76:12181 ls /brokers/topics`

Note: Replace the IP address and port (that is, 129.40.23.76:12181) with information about any other ZooKeeper instance (129.40.23.76:12181, 129.40.23.76:22181, or 129.40.23.76:32181).

Another helpful command is **docker logs**, which provides information that is logged by a running container.

2.8 Integrating IBM Z applications and Kafka

In this section, we describe different types of applications and their integration with Kafka.

2.8.1 Event-driven application and Apache Kafka

Competition and disruptors are causing companies to become more customer-centric. This move led to a surge in event-driven solutions and applications. Event driven solutions require different thinking than data-centric solutions. Events form the nervous system of the digital business

Application infrastructure must provide event stream processing capabilities and support emerging event-driven programming models. This event-driven journey and underpins the next generation of digital customer experiences.

The components of an event driven application are shown in Figure 2-4.

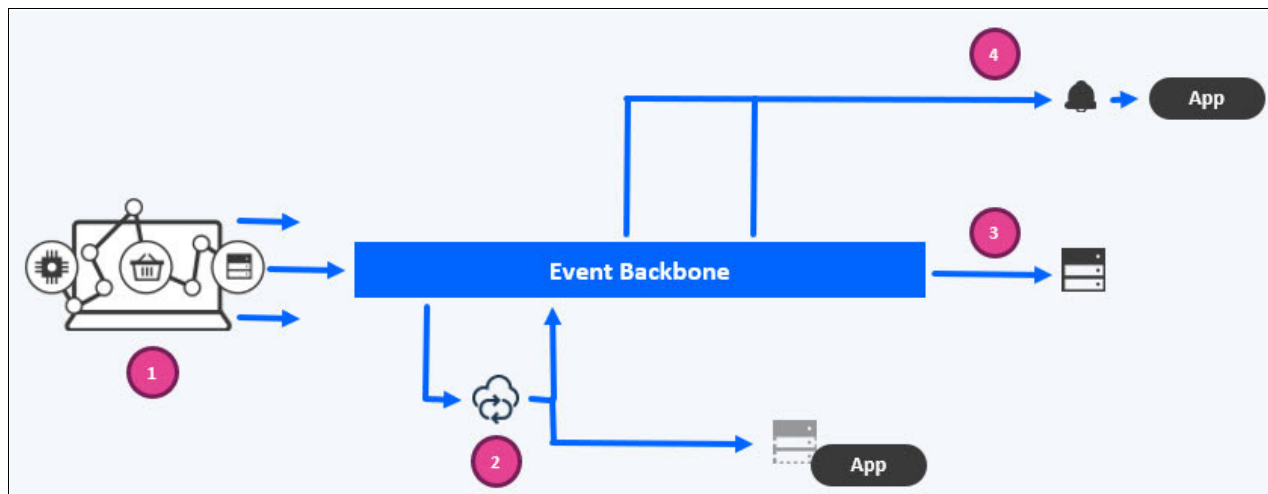


Figure 2-4 Components of event driven application

An event-driven application features the following components (as numbered in Figure 2-4):

1. *Event sources* are application triggers, data in databases, sensors, web traffic, and so on. Event sources can also be transactions on the mainframe that we discuss in our use case in 2.10, “Sending CICS events to Kafka” on page 74. These event sources publish event data to the event backbone.
2. *Stream processing* are applications that process streams of data that are in the event backbone in real time.
3. *Event archives* are applications or tools that store the events from the event backbone for future reference.

4. *Notifications* are applications that trigger notifications based on some events that are stored in the event backbone.

These four components are connected by an event backbone. Apache Kafka is a perfect match for this event backbone.

2.8.2 Key usage patterns of Apache Kafka and IBM Z

Companies are facing growing competition and want to modernize their customer experience to improve customer retention. Companies run their mission-critical business applications and store their data on the mainframe.

Companies want to build and offer services to customers who can be alerted in real-time when “noteworthy” events occur. Those events vary by applications and industries.

Customers do not want to disrupt or add load to their core systems that handle millions of transactions a day and are costly and complex to change. To stay competitive, companies want to unlock the data that is flowing through its transactional systems.

In this section, we examine one of the key usage pattern of unlocking events from the mainframe by using Apache Kafka.

One of the most common usage patterns for Kafka with IBM Z is shown in Figure 2-5.

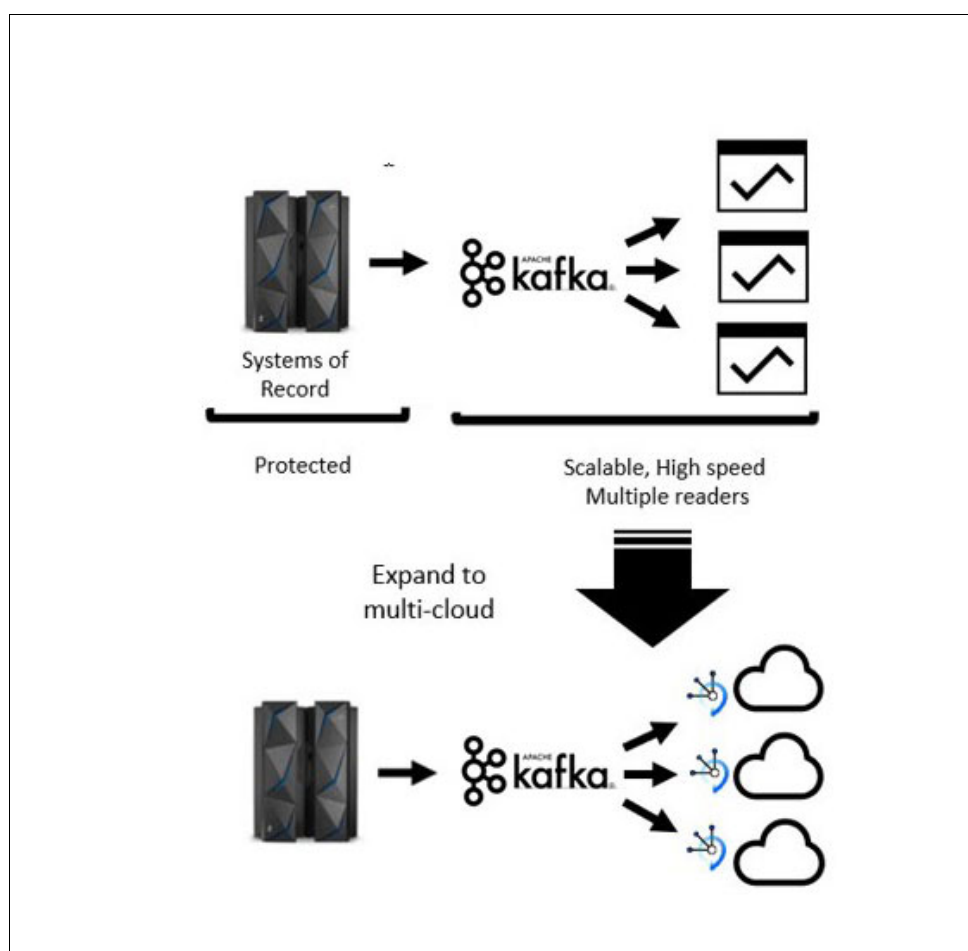


Figure 2-5 Key Usage Pattern

Multiple teams that are responsible for innovation, applications, and microservices are demanding data from the systems of record, which is often hosted on an IBM Z. The teams that are running the systems of record must ensure that they are stable, but this new demand is unpredictable, and often not well-defined at the outset. Kafka offers a way to make the data available so that it can be used by multiple readers at their own pace. The load on the backend systems is well-defined and predictable.

Kafka offers the data as a persistent read-only buffer or cache of the data, which is continually refreshed and updated.

We can progress that scenario to the next level by pushing that data out to multiple users on multiple clouds. Allowing each set of users on each cloud to have its own buffer or stream of the data allows them to have a consistent, rapid read of the data while minimizing the amount of data that is transferred.

2.8.3 Unlocking data from IBM Z to Kafka

Companies run their mission-critical core transactions on the mainframe with CICS, IMS, and IBM Db2®. We can unlock the data on those transactional systems and databases by using one of the various options that are shown in Figure 2-6.

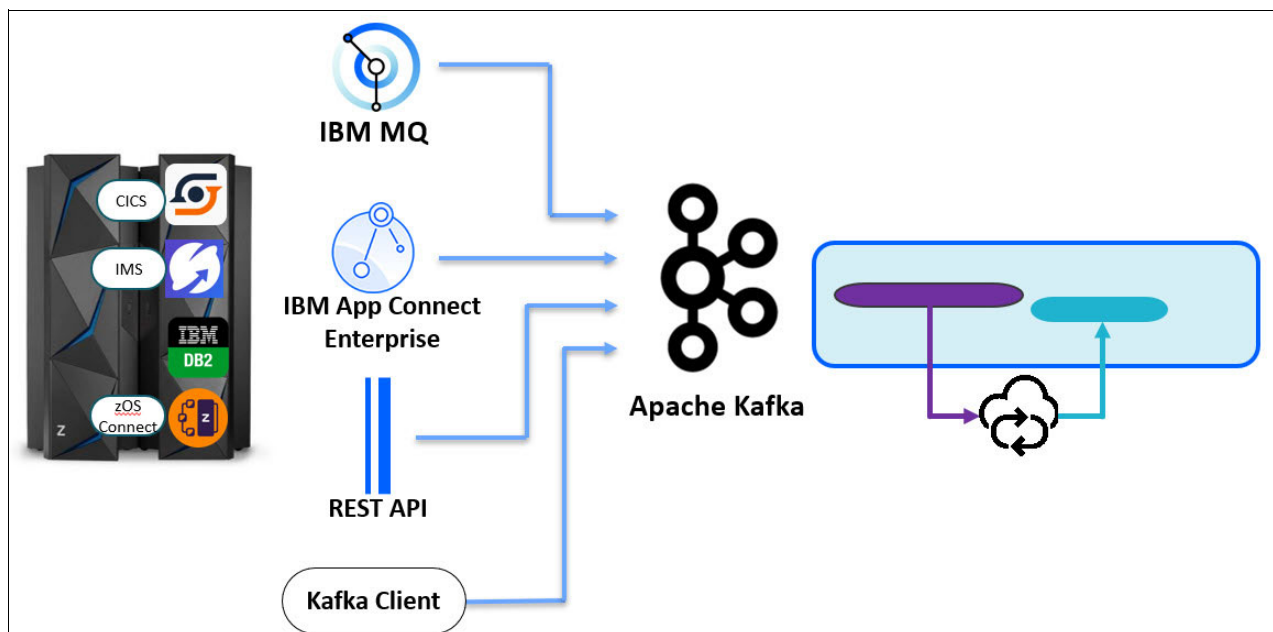


Figure 2-6 Integration to Kafka from IBM Z

The following products can integrate with Apache Kafka:

- ▶ IBM MQ: Use of the IBM MQ Kafka Connectors
- ▶ IBM App Connect Enterprise: Use of the Kafka nodes
- ▶ IBM z/OS Connect EE: Outbound REST API call to Kafka
- ▶ CICS: Using Liberty Kafka client running in CICS
- ▶ IBM InfoSphere® Data Replication - Change Data Capture

2.9 Integration of IBM MQ with Apache Kafka

Many organizations use IBM MQ and Apache Kafka for their messaging needs. Although they are generally used to solve different types of messaging problems, users often want to connect them for various reasons. For example, IBM MQ can be integrated with systems of record while Apache Kafka is commonly used for streaming events from web applications.

The ability to connect the two systems enables scenarios in which these two environments intersect. It is easy to connect from IBM MQ to Apache Kafka. IBM created a pair of connectors, which are available as source code or as part of IBM Event Streams. The following connectors are available:

- Kafka Connect source connector for IBM MQ

Kafka-connect-mq-source is a Kafka Connect source connector for copying data from IBM MQ into Apache Kafka. For more information about the IBM MQ source connector, see this [web page](#).

- Kafka Connect sink connector for IBM MQ

You can use the IBM MQ sink connector to copy data from Apache Kafka into IBM MQ. The connector copies messages from a Kafka topic into an IBM MQ queue. For more information about the IBM MQ sink connector, see this [web page](#).

In this section, we discuss the popular IBM MQ connectors for integration with Apache Kafka.

2.9.1 Setting up IBM MQ connectors to run on IBM z/OS

In this section, we describe how to build and run the IBM MQ connectors. The connectors can be deployed and run in a Kafka Connect runtime; for example, in z/OS UNIX System Services. In this section, we build and configure the connectors that are running on z/OS UNIX System Services, as shown in Figure 2-7.

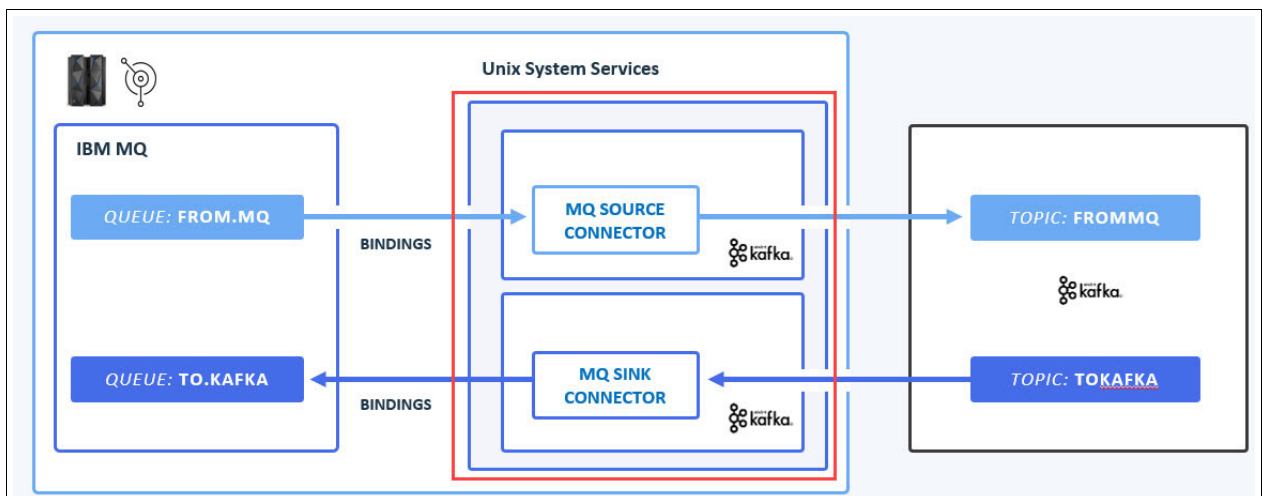


Figure 2-7 Running connectors on UNIX System Services

Complete the following steps to set up the connectors on IBM z/OS UNIX System Services:

1. Set up Apache Kafka to run on IBM z/OS:
 - a. Download Apache Kafka 2.0.0 or later to a non-z/OS system at this [web page](#).

- b. On the system that you downloaded the file, extract the downloaded .tgz file to a .tar by using gunzip, as shown in Example 2-52.

Example 2-52 Extract to tar file

```
gunzip -k kafka_2.12-2.5.0.tgz
```

- c. Transfer the resulting kafka_2.12-2.5.0.tar file to the /u/maitra/kafkadown1/ directory on z/OS UNIX System Services, as shown in Example 2-53.

Example 2-53 Transfer the tar file

```
MAITRA@SC59:/u/maitra/kafkadown1> ls kafka*  
kafka_2.12-2.5.0.tar
```

2. Extract the Apache Kafka distribution:

- a. Create a directory and browse to the new directory, as shown in Example 2-54.

Example 2-54 Create directory for kafka

```
MAITRA@SC59:/u/maitra>  
==> mkdir kafka  
MAITRA@SC59:/u/maitra> cd kafka  
MAITRA@SC59:/u/maitra/kafka>
```

- b. Copy the kafka_2.12-2.5.0.tar file from /u/maitra/kafkadown1/ to the new directory.
- c. Extract the kafka_2.12-2.5.0.tar file, as shown in Example 2-55.

Example 2-55 Extract tar file

```
tar -xvf kafka_2.12-2.5.0.tar
```

- d. Browse to the newly created directory, as shown in Example 2-56.

Example 2-56

```
MAITRA@SC59:/u/maitra/kafka>  
==> cd kafka_2.12-2.5.0
```

3. Convert the shell scripts to run in z/OS UNIX System Services:

- a. We are going to run the connectors in distributed mode and perform a distributed setup. Copy the connect-distributed.sh shell script into the current directory, as shown in Example 2-57.

Example 2-57 Copy connect-distributed.sh

```
MAITRA@SC59:/u/maitra/kafka/kafka_2.12-2.5.0>  
==>cp bin/connect-distributed.sh ./connect-distributed.sh.orig
```

- b. Determine the code set on the IBM z/OS system by running the locale command, as shown in Example 2-58.

Example 2-58

```
MAITRA@SC59:/u/maitra/kafka1/kafka_2.12-2.2.0> locale -k codeset  
codeset="IBM-1047"
```

- c. Convert the script to EBCDIC encoding and replace the original for code set IBM-1047 by using the command that is shown in Example 2-59.

Example 2-59

```
iconv -f ISO8859-1 -t IBM-1047 ./connect-distributed.sh.orig >  
bin/connect-distributed.sh
```

- d. Ensure the file permissions are set so that the script is executable by running the command that is shown in Example 2-60.

Example 2-60 Change file permission for distributed.sh

```
chmod +x bin/connect-distributed.sh
```

- e. Convert the kafka-run-class.sh shell script by copying the kafka-run-class.sh shell script into the current directory, as shown in Example 2-61.

Example 2-61 Copy the shell script

```
cp bin/kafka-run-class.sh ./kafka-run-class.sh.orig
```

- f. Convert the script to EBCDIC encoding and replace the original for code set IBM-1047 by using the command that is shown in Example 2-62.

Example 2-62 Convert kafka-run-class.sh

```
iconv -f ISO8859-1 -t IBM-1047 ./kafka-run-class.sh.orig >  
bin/kafka-run-class.sh
```

- g. Ensure the file permissions are set so that the script is executable by running the command that is shown in Example 2-63.

Example 2-63 Change permissions

```
chmod +x bin/kafka-run-class.sh
```

- 4. Download IBM MQ Source connectors and configuration files.

In our use example, we download and configure the IBM MQ Source Connector. If you are licensed to run IBM Event Streams, you can download the connector from the IBM Event Streams UI by completing the following steps:

- a. Log in to the IBM Event Streams UI from a supported web browser.
- b. Click **Toolbox** in the primary window, as shown in Figure 2-8 on page 67.

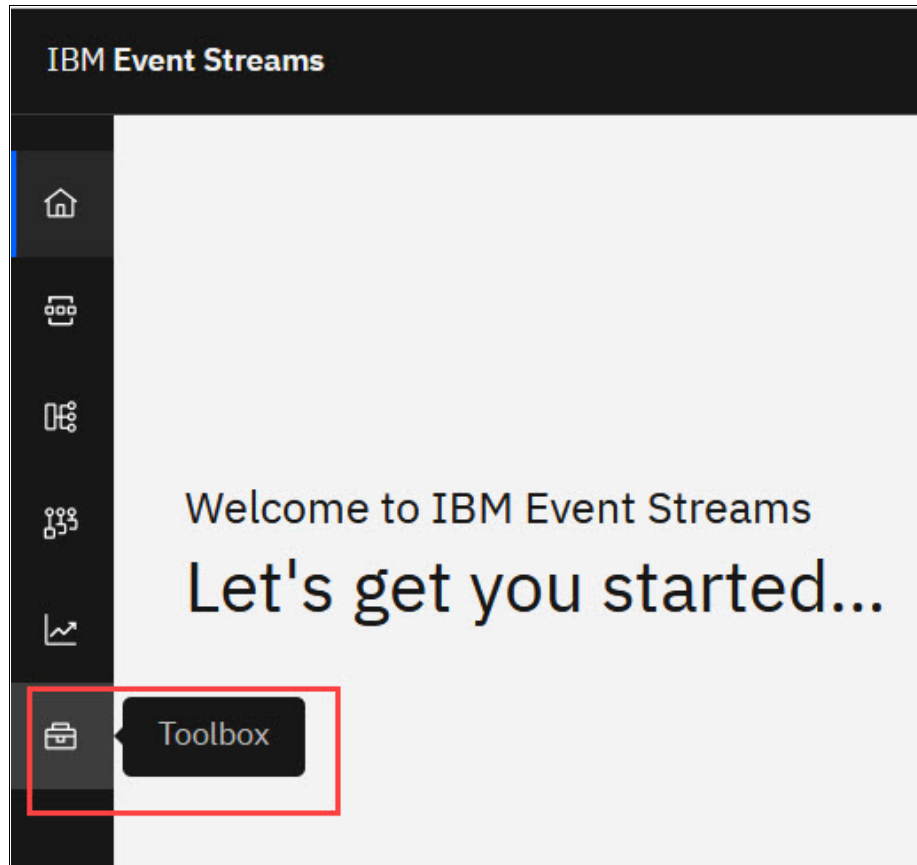


Figure 2-8 IBM Event Streams UI Toolbox

- c. Scroll to the Connectors section and click **Connecting to IBM MQ**, as shown in Figure 2-9

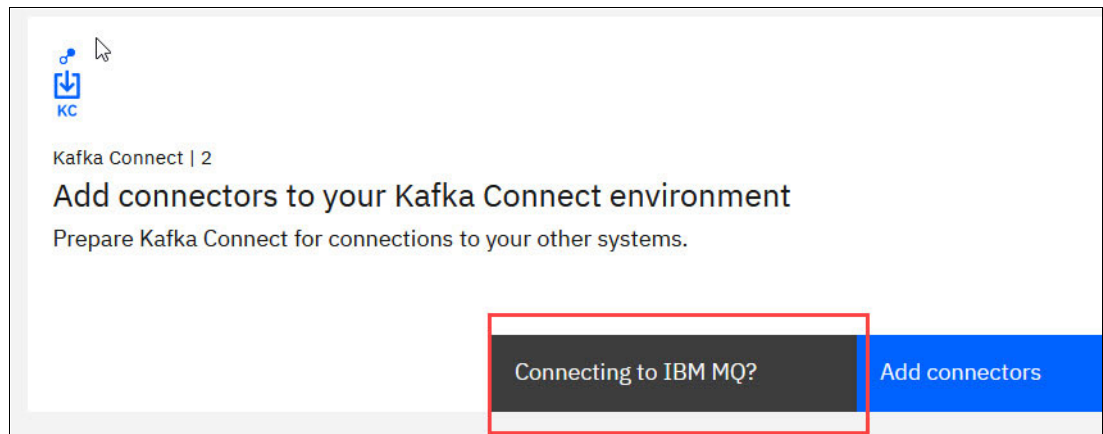


Figure 2-9 Get IBM MQ connector and configuration files

- d. Ensure that the IBM MQ Source tab is selected and download the connector file and configuration file, as shown in Figure 2-10 on page 68. Two files are downloaded to the local system and must be transferred to z/OS UNIX System Services:
 - kafka-connect-mq-source-1.1.1-jar-with-dependencies.jar
 - mq-source.json

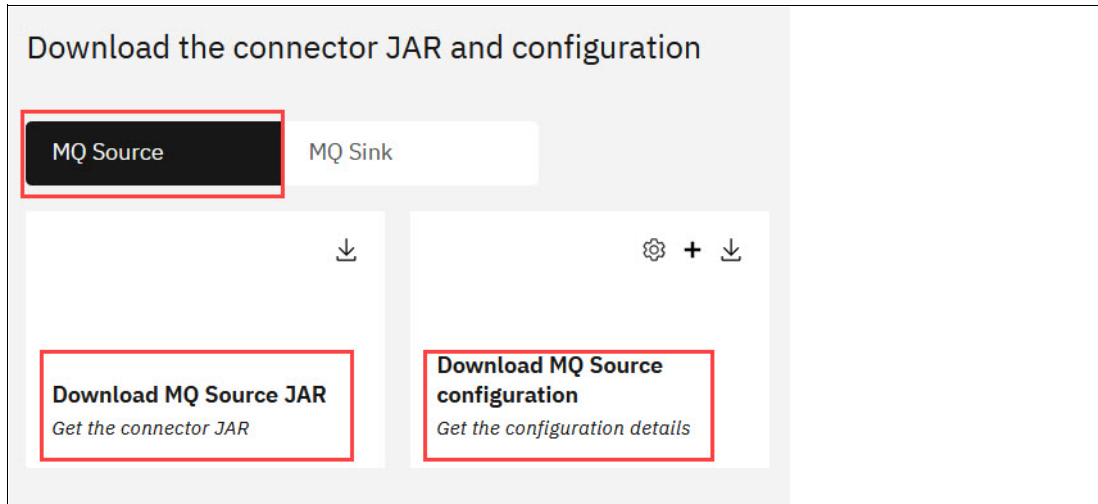


Figure 2-10 Download connector and configuration files

- e. Edit the `mq-source.json` file and the content should look as shown in Example 2-64. Transfer the `mq-source.json` in binary format to z/OS UNIX System Services to the following directory `/u/maitra/kafka/kafka_2.12-2.5.0`. The `mq-source.json` file should remain in ASCII format.

Example 2-64 mq-source.json

```
{
  "name": "mq-source",
  "config": {

    "connector.class": "com.ibm.eventstreams.connect.mqsource.MQSourceConnector",
    "tasks.max": "1",
    "mq.queue.manager": "MQR1",
    "mq.connection.mode": "bindings",
    "mq.queue": "FROM.MQ",

    "mq.record.builder": "com.ibm.eventstreams.connect.mqsource.builders.DefaultRecordBuilder",
    "topic": "fromMQ",

    "key.converter": "org.apache.kafka.connect.storage.StringConverter",

    "value.converter": "org.apache.kafka.connect.converters.ByteArrayConverter"
  }
}
```

- f. Transfer the `.jar` file to z/OS UNIX System Services to the following directory: `/u/maitra/kafka/kafka_2.12-2.5.0`.
- g. If you are not entitled to IBM Event Streams, you can follow the steps that are outlined in the following web pages to build the `.jar` file and get the sample configuration file:
 - <https://github.com/ibm-messaging/kafka-connect-mq-source#building-the-connector>
 - <https://github.com/ibm-messaging/kafka-connect-mq-source/tree/master/config>

5. Convert the properties file to EBCDIC format to run in z/OS UNIX System Services:
 - a. Copy the connect-distributed.properties file into the current directory, as shown in Example 2-65.

Example 2-65 Copy connect-distributed.properties

```
cp config/connect-distributed.properties
./connect-distributed.properties.orig
```

- b. Convert the properties file to EBCDIC encoding and replace the original, as shown in Example 2-66.

Example 2-66 Convert property file to EBCDIC

```
iconv -f ISO8859-1 -t IBM-1047 ./connect-distributed.properties.orig >
config/connect-distributed.properties
```

6. Update the Kafka Connect configuration.

The connect-distributed.properties file must include the correct bootstrap.server for your Apache Kafka install. Our bootstrap server is 129.40.23.76:19092. Update the connect-distributed.properties file, as shown in Example 2-67.

Example 2-67 Updated connect-distributed.properties

```
bootstrap.servers=129.40.23.76:19092
group.id=connect-cluster
key.converter=org.apache.kafka.connect.json.JsonConverter
value.converter=org.apache.kafka.connect.json.JsonConverter
key.converter.schemas.enable=true
value.converter.schemas.enable=true
offset.flush.interval.ms=10000
offset.storage.topic=connect-offsets
offset.storage.replication.factor=3
offset.storage.partitions=25
status.storage.topic=connect-status
status.storage.replication.factor=3
status.storage.partitions=5
config.storage.topic=connect-configs
config.storage.replication.factor=3
```

7. Configure the environment to run the connector.

The IBM MQ connectors use the JMS API to connect to IBM MQ. We must set the required environment variables for JMS applications before running the connectors on IBM z/OS.

We also must ensure that we set CLASSPATH to include com.ibm.mq.allclient.jar and set the .jar file for the connector we use. This connector is the .jar file that we downloaded from the Event Streams UI or built after cloning the GitHub project, for example, kafka-connect-mq-source-1.1.1-jar-with-dependencies.jar, as shown in Example 2-68.

Example 2-68 Set CLASSPATH

```
CP1=/usr/lpp/mqm/V9R1M0/java/lib/com.ibm.mq.allclient.jar
CP2=/u/maitra/kafkadown1/kafka-connect-mq-source-1.1.1-jar-with-dependencies.jar
CP3=/u/maitra/kafkadown1/kafka-connect-mq-sink-1.1.1-jar-with-dependencies.jar
```

```
export CLASSPATH=$CP1:$CP2:$CP3
```

We use the bindings connection mode for the connector to connect to the queue manager; therefore, you must also set the following environment variables:

- The STEPLIB that is used at run time must contain the IBM MQ SCSQAUTH and SCSQANLE libraries. Specify this library by using the .profile file.

We add the IBM MQ SCSQAUTH and SCSQANLE libraries by using a line in our .profile file, as shown in Example 2-69, replacing the high level qualifier with the high-level data set qualifier that we chose when installing IBM MQ, as shown in Example 2-69.

Example 2-69 Add IBM MQ STEPLIB

```
export STEPLIB=MQ910.SCSQAUTH:MQ910.SCSQANLE
```

- The connector needs to load a native library. Set LIBPATH to include the directory of the IBM MQ installation, as shown in Example 2-70.

Example 2-70 Set LIBPATH

```
L1=/usr/lpp/mqm/V9R1M0/java/lib  
export LIBPATH=$L1
```

2.9.2 Starting Kafka Connect on z/OS

Kafka Connect is started by using a bash script. To run the Kafka connect in distributed mode, complete the following steps:

1. Start Kafka Connect in distributed mode:
 - a. Browse to the Kafka directory and run the connect-distributed.sh script, passing in the connect-distributed.properties file, as shown in Example 2-71.

Example 2-71 Run connect-distributed

```
cd /u/maitra/kafka/kafka_2.12-2.5.0  
./bin/connect-distributed.sh connect-distributed.properties
```

- b. Start an individual mq-source connector by using the Kafka Connect REST API, as shown in Example 2-72.

Example 2-72 Start source connector

```
curl -X POST -H "Content-Type: application/json"  
http://localhost:8083/connectors -d @mq-source.json
```

2.9.3 Status of plug-ins and connectors

To see whether the IBM MQ connectors are installed properly, open a browser and enter the following URL with your own host name:

<http://wtsc59.pbm.ihost.com:8083/connector-plugins>

You see the components that are installed, as shown in Example 2-73, which indicates that the connectors are installed correctly.

Example 2-73 Installed plug-ins

```
0:
class: com.ibm.eventstreams.connect.mqsink.MQSinkConnector"
type: sink"
version: 1.1.1"
1:
class"com.ibm.eventstreams.connect.mqsource.MQSourceConnector"
type"source"
version"1.1.1"
2:
class"org.apache.kafka.connect.file.FileStreamSinkConnector"
type"sink"
version"2.5.0"
3:
class"org.apache.kafka.connect.file.FileStreamSourceConnector"
type"source"
version"2.5.0"
```

To see the status of the mq-source connector, open the browser and enter the following URL with your own host name:

<http://wtsc59.pbm.ihost.com/connectors/mq-source/status>

You see the status of the mq-source connector, as shown in Figure 2-11.



Figure 2-11 Status of IBM MQ Source Connector

2.9.4 Sending messages from IBM MQ to Kafka

In the previous section, we set up the IBM MQ connectors on z/OS UNIX System Services and started the mq-source connector.

Next, we send messages from an IBM MQ Queue on z/OS to a Kafka topic on zCX. The infrastructure that is shown in Figure 2-12 on page 72 is in place.

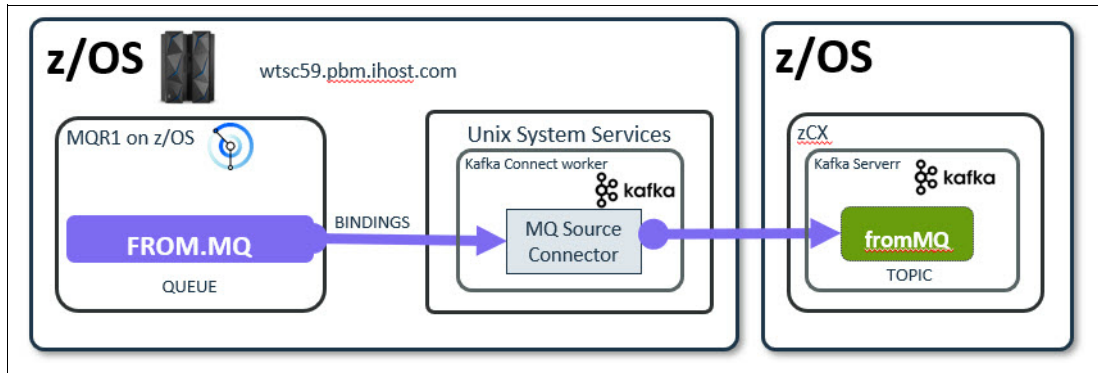


Figure 2-12 Sensing messages from IBM MQ to Kafka

The following components are running:

- ▶ Queue Manager on z/OS: MQR1
- ▶ Local Queue on MQR1: FROM.MQ
- ▶ MQ-Source connector worker running on UNIX System Services
- ▶ Kafka Cluster running on zCX: bootstrap server -> 129.40.23.76:19092
- ▶ Kafka Topic: fromMQ

We put a message on queue FROM.MQ on Queue Manager MQR1 by using the IBM MQ Explorer tool, as shown in Figure 2-13.

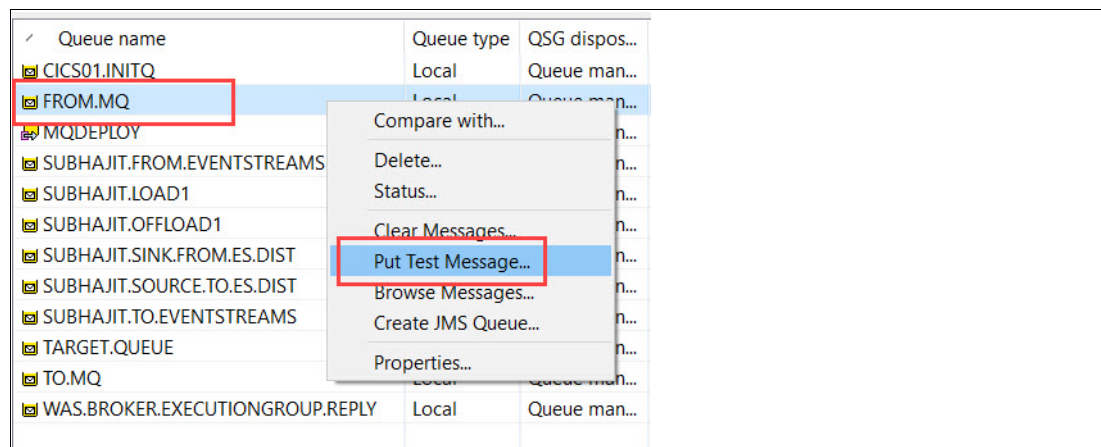


Figure 2-13 Put message on Queue

We then enter the message text, as shown in Figure 2-14 on page 73.

Put test message

Put message to:

Queue manager:

MQR1

Queue:

FROM.MQ

Message data:

This message if from MQR1 on zOS to Kafka Topic - msg1

Put message Close

Figure 2-14 Content of message

Next, we check the current depth of the queue FROM.MQ. We see that the depth of the queue is 0. This depth means that the mq-source connector used the message, as shown in Figure 2-15.

Queue name	Queue type	QSG dispos...	Open input count	Open output count	Current que...
QCS01.INITQ	Local	Queue man...	0	0	0
FROM.MQ	Local	Queue man...	1	0	0
MQDEPLOY	Local	Queue man...	0	0	0

Figure 2-15 Current depth of queue

If the mq-source connector worked correctly, we see the message in the Kafka topic from IBM MQ on the Kafka cluster running on zCX. We use the `kafkacat` utility to browse the content of the topic. Log on to zCX and run the command that is shown in Example 2-74.

Example 2-74 Browse stream data on topic

```
docker exec -it kafka-app-1 bash -c "kafkacat -b 129.40.23.76:19092 -t fromMQ"
```

The output of the command is shown in Example 2-75. It shows that the message we put in the queue FROM.MQ landed on the Kafka topic from IBM MQ. Our connector test is complete.

Example 2-75 Contents of topic

```
admin@sc74cn09:~$ docker exec -it kafka-app-1 bash -c "kafkacat -b
129.40.23.76:19092 -t fromMQ"
% Auto-selecting Consumer mode (use -P or -C to override)
hello world
This message if from MQR1 on zOS to Kafka Topic - msg1
```

For more information about any of the topics found in this chapter, see “Related publications” on page 255.

2.10 Sending CICS events to Kafka

In this section, we describe how CICS events can be sent to Kafka servers running in zCX.

2.10.1 Why CICS events?

CICS is used by many businesses to run core applications. When these applications are run, many events occur that can be of interest to other parts of the business. For example, a business might want to send an SMS message to a customer when a large amount of money is withdrawn from their bank account.

One way to achieve that goal is to modify the program, test it, and then implement it in production. However, this process can take some time to implement and require numerous testing.

CICS events provides a way to quickly implement the capture of an event and its subsequent processing without requiring any program changes. The advantage is that implementing a CICS event can be done far quicker than a comparable program change with less risk of disruption.

For more information about CICS events, see the following resources:

- ▶ [IBM Knowledge Center](#)
- ▶ *Event Processing with CICS*, SG24-7792

2.10.2 CICS to Kafka overview

Figure 2-16 on page 75 shows the configuration we set up to demonstrate the use of CICS events to capture a business event and sending the captured data to a topic in Kafka.

We used a CICS sample application to demonstrate sending a CICS event to Kafka in zCX. A copy of this example is available by downloading the `CicsToKafkaProjects.zip` file. For more information about instructions to download this file, see Appendix A, “Additional material” on page 253.

The file contains the following projects:

- ▶ `CatalogEvents`: The CICS event
- ▶ `CicsToKafkaDemo`: The ZKAFKA Java program
- ▶ `CicsToKafkaBundle`: Bundle definition that us used to install the ZKAFKA Java program

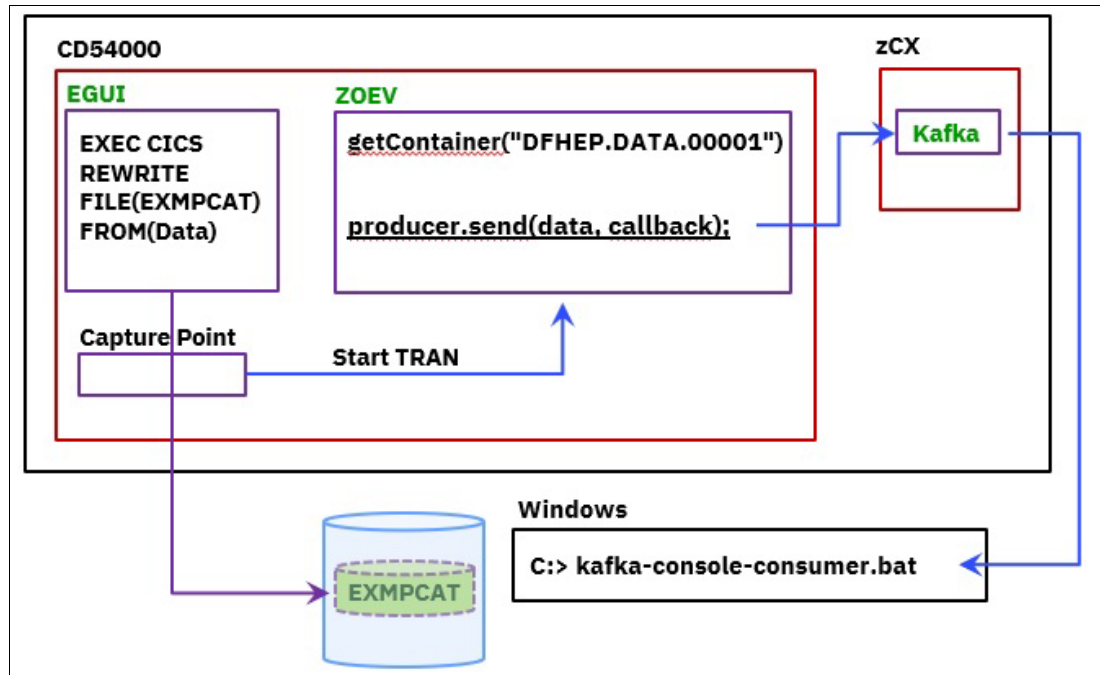


Figure 2-16 CICS to Kafka configuration

Flow of the example

The following steps explain how our example functions. Each component in this flow is described in 2.10.3, “Example components” on page 76:

1. The CICS transaction EGUI is run in a 3270 screen.
2. The display that is shown in Figure 2-17 shows an order being placed.

```

CICS EXAMPLE CATALOG APPLICATION - Details of your order

Enter order details, then press ENTER

Item      Description                                Cost      Stock      On
-----
0010      Ball Pens Black 24pk                      2.90      0109

Order Quantity: 1
User Name: Edward
Charge Dept: IBM Z
  
```

Figure 2-17 Placing an order

3. When Enter is pressed, the program issues the following CICS API:
EXEC CICS REWRITE FILE(EXMPCAT) FROM(Data)
4. The CICS event detects the API and at the capture point it collects data from that event. In this case, it collects all the updated data that is written to the VSAM file that is called EXMPCAT.
5. The CICS EGUI transaction completes as normal.

6. The CICS event starts a new CICS transaction that is called ZOEV and passes the collected data by way of a CICS container.
7. The started transaction starts a Java program that is called ZKAFKA, which is a Java program that is running in a Liberty server in CICS.
8. The Java program reads the data from the CICS container and then sends it to a topic called fromCICSEVENT in the Kafka server that is running in zCX.
9. A Kafka client program is running on Microsoft Windows is used to echo the message that was sent to the Kafka topic by CICS.

2.10.3 Example components

In this section, we provide an overview of the key components that we used to set up this example.

CICS sample application

CICS supplies a sample application that is called the Catalog Manager application. For more information about this application and how to set it up in CICS, see [IBM Knowledge Center](#).

We set up this application in a CICS region that is called CD54000.

The application is started by using a transaction that is called EGUI, which is supplied in the sample Catalog Manager application.

CICS event bundle

We used the sample of the CICS event to capture when the Catalog Manager Application updates the EXMPCAT file. The update occurs when an order is placed by using the 3270 screen. The event traps on the program that is running the following command:

```
EXEC CICS REWRITE FILE(EXMPCAT) FROM(Data)
```

When the CICS Event is triggered by this command, it is configured to start a CICS transaction that is named ZOEV that is running a program that is named ZKAFKA.

CICS Liberty JVM

We configured a Liberty server in the CICS region.

For more information about how to configure a Liberty server in CICS, see [IBM Knowledge Center](#).

We named the server that we created WLP54000. The Liberty server was configured to use SAF as the authentication mechanism, which requires an associated Liberty Angel process to be running.

ZKAFKA Java program

A Java program that is named ZKAFKA runs in a Liberty server in CICS. It reads the data passed to it in CICS containers and sends the data to the Kafka server running in zCX. For more information about this program, see 2.10.4, “ZKAFKA Java program” on page 77.

Kafka server in zCX

We used the Kafka cluster running in zCX that was set up as described in 2.5.1, “Creating the Kafka cluster” on page 18.

Windows Kafka client

We used a Kafka client program to subscribe to the Kafka topic that displays the messages that are written to Kafka by the ZKAFKA program in CICS. For more information about how to download this client, see “Using a Kafka client program” on page 79.

2.10.4 ZKAFKA Java program

We used a Java program that is named ZKAFKA that runs in a Liberty server in CICS to send the data that is captured by the CICS event to the Kafka server running in zCX.

Example 2-76 shows the Java code that is used to obtain the data that is passed in the CICS containers.

Example 2-76 Java code to read contents of the CICS Containers

```
try {
    ContainerIterator ci2 = eventChlData.containerIterator();
    while (ci2.hasNext()) {
        Container contId = ci2.next();
        System.out.println(cTime + ">>>cont: " + contId.getName());
        if (!contId.getName().startsWith("DFHEP.CCECONTEXT")) {
            if (contId.getName().startsWith("DFHEP.NAME")) {
                if (eventData.length() > 0)
                    eventData = eventData + ",";
                eventData = eventData + contId.getString().trim() + ":";
            } else if (contId.getName().startsWith("DFHEP.DATA"))
                eventData = eventData + contId.getString().trim();
        }
    }
}
```

Example 2-77 shows the Java code that is used to send data to Topic in Kafka.

Example 2-77 Java code to send data to Topic in Kafka

```
// Specify TCP/IP address and port where Kafka server is located
String KAFKA_BROKER_LIST = "129.40.23.77:19092";
String CLIENT_ID = "zzz";
// Specify name of Kafka topic to send to
String TOPIC = "fromCICSEVENT"; Properties properties = new Properties();

properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, KAFKA_BROKER_LIST);
properties.put(ProducerConfig.CLIENT_ID_CONFIG, CLIENT_ID);
properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringSerializer");
properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
    "org.apache.kafka.common.serialization.StringSerializer");

KafkaProducer producer = new KafkaProducer<>(properties);
ProducerRecord data = new ProducerRecord(TOPIC, eventData);

// Set up for call back after data has been sent to Kafka

TestCallback callback = new TestCallback();

// Send the data to Kafka

producer.send(data, callback);
```

We coded the Java program to write several messages to the Liberty server STDOUT log.

2.10.5 Testing the example

Figure 2-17 on page 75 showed how we ran the CICS Catalog Sample Application to place an order.

When Enter was pressed in the EGUI transaction, the output from the ZKAFKA Java program was shown in the STDOUT log and is shown in Example 2-78. On our system, the STDOUT log for the Liberty server running in CICS is in the following directory:

/u/edmcarr/CD54000/WLP54000/CURRENT.STDOUT

Example 2-78 Output from ZKAFKA Java program

```
2020/07/22 23:01:21>>>cont: DFHEP.CCECONTEXT
2020/07/22 23:01:21>>>cont: DFHEP.NAME.00001
2020/07/22 23:01:21>>>cont: DFHEP.DATA.00001
2020/07/22 23:01:21>>>cont: DFHEP.NAME.00002
2020/07/22 23:01:21>>>cont: DFHEP.DATA.00002
2020/07/22 23:01:21>>>cont: DFHEP.NAME.00003
2020/07/22 23:01:21>>>cont: DFHEP.DATA.00003
2020/07/22 23:01:21>>>cont: DFHEP.NAME.00004
2020/07/22 23:01:21>>>cont: DFHEP.DATA.00004
2020/07/22 23:01:21>>>cont: DFHEP.NAME.00005
2020/07/22 23:01:21>>>cont: DFHEP.DATA.00005
2020/07/22 23:01:21>>>Data sending to Kafka:
Program_name:DFH0XVDS,Item_ref:0080,Item_description:Laser Paper 28-lb 108 Bright
2500/case,in_stock:0024,on_order:000
2020/07/22 23:01:21>>>Time for first send of data of length: 0 to Kafka: 381ms
2020/07/22 23:01:21>>>Time for second send of data of length: 0 to Kafka: 0ns
2020/07/22 23:01:21>>>CICS sent message to topic: fromCICSEVENT partition:0
offset:15
2020/07/22 23:01:21>>>CICS sent message to topic: fromCICSEVENT partition:0
offset:16
```

We set up the ZKAFKA Java program to send the data to the Topic in Kafka twice.

On our system, the first send of the data took 381ms. This result is to be expected because the first send requires classes to be loaded, a TCPIP connection to be established between CICS, and the Kafka server running in ZCX.

When the ZKAFKA Java program sends the data a second time, the TCPIP connection was already established and classes loaded, so the second send took less time. In fact, it showed as zero nanoseconds.

In a real production environment, you can consider setting up an Object Pool design pattern.

The approach is to create an object pool of KafkaProducer objects, with these objects being in effect already created connections to the Kafka server. The Java program can get a Kafka Producer object from the pool, use it, and then return it to the pool.

This process saves the Java program from having to create a connection to the Kafka server on each call and improves performance.

One open source project that can be used for this purpose is the Apache Commons Pool project, which can be found in the `org.apache.commons.pool2` package.

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

Using a Kafka client program

To run more tests, we downloaded a Kafka client Java program to an MS Windows PC from this [web page](#).

Under the release heading 2.4.1, we selected the Scala 2.12 file that is named `kafka_2.12-2.4.1.tgz` and downloaded it to our PC.

We unzipped the file to a directory that is named `C:\kafka_2.11-2.4.1`. We found that Kafka required a short directory name to function correctly on Windows.

The subdirectory that is named `libs` contains Kafka JAR files that are needed for the Java program.

We installed the Kafka product code and ran a supplied Kafka consumer by running the following command:

```
kafka-console-consumer.bat --bootstrap-server 129.40.23.77:19092 --topic  
fromCICSEVENT --from-beginning
```

When we placed the order in Windows, we saw the output that is shown in Example 2-79.

Example 2-79 Data from topic in Kafka

EPFE0002z0ffice0rdersEvents	z0fficeOrder
1910E4E2C9C2D4E2C34BE3C3D7F5F5F0F2F9380F7BF57AE2000100USIBMSC.CD54000	
2020-07-15T01:17:12.803+00:00z0fficeOrderPlaced	
OrderData	0030Ball Pens Red 24pk
010002.900105000	

This output showed that the CICS event captured the updated data, and that the Java program in CICS successfully sent that data to the Kafka server.

2.10.6 z/OS Explorer

We used z/OS Explorer to develop the ZKAFKA Java program. The z/OS Explorer can be downloaded from this [web page](#).

Figure 2-18 shows the download window for z/OS Explorer.

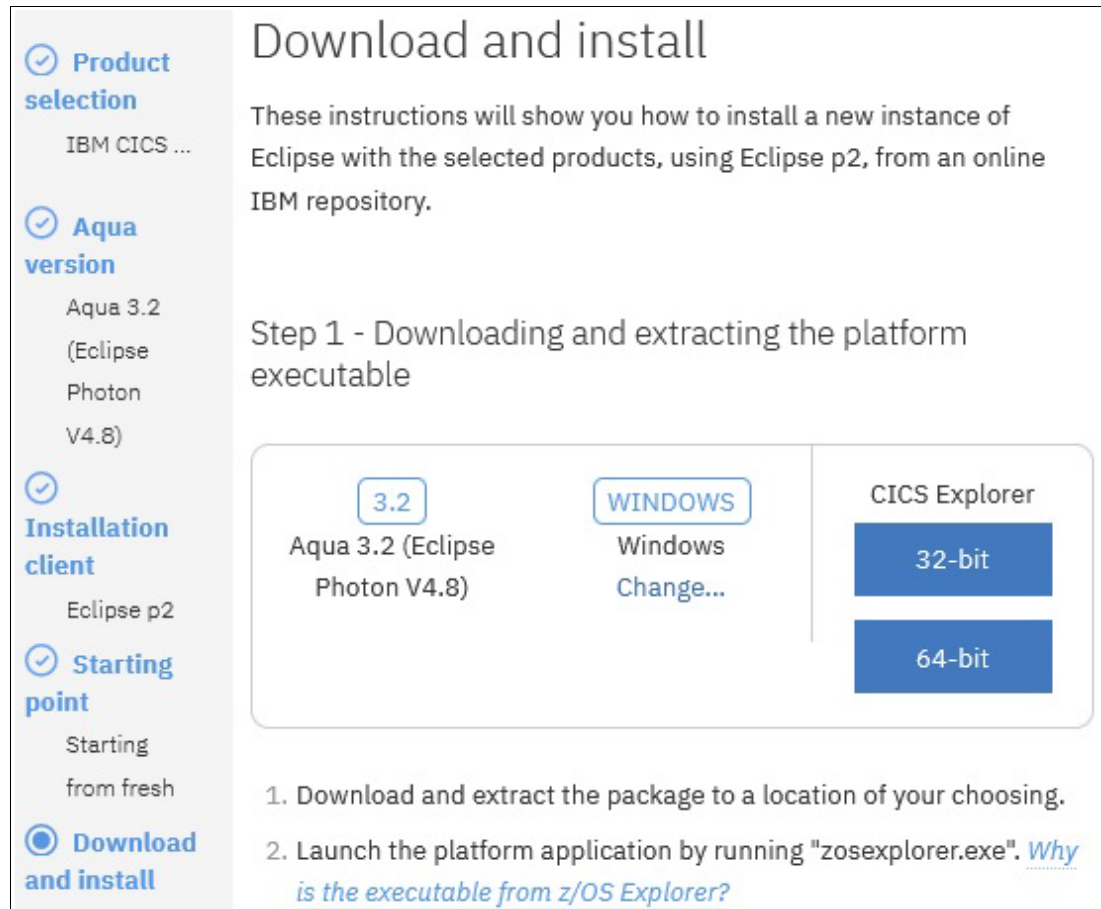


Figure 2-18 Downloading z/OS Explorer

We performed the following steps:

1. We clicked the **64-bit** button after which we were presented with a display prompting us to agree to the licensing agreement. After accepting the agreement, a file that is named cicsexplorer-aqua-5.5.9-win32-x86_64.zip (601 MB) was downloaded to our PC.
2. We decompressed this file into a parent directory that is named C:\zProducts\zosExp\.
3. From the C:\zProducts\zosExp\IBM Explorer directory for z/OS, we started the program zosexplorer.exe.

4. After z/OS Explorer started, we selected **Help**, then '**Install New Software...**'. In the displayed window, we selected the drop-down for '**Work with:**' and selected the entry that starts with IBM Explorer for z/OS Update Site, as shown in Figure 2-19.

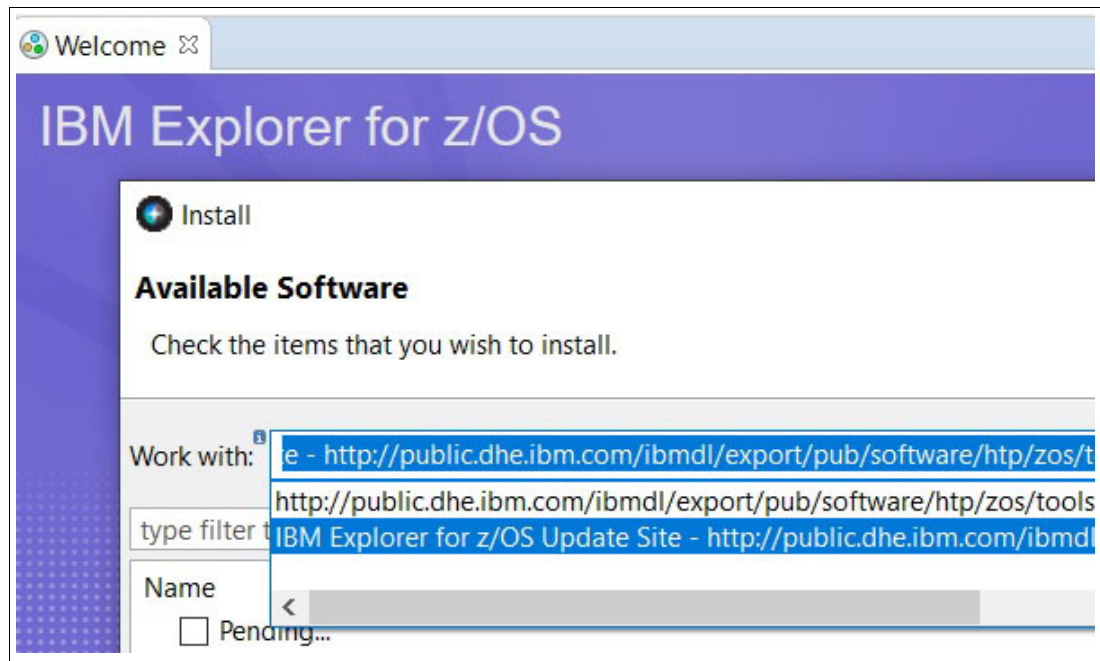


Figure 2-19 Selecting update site

5. We selected **CICS Explorer®** from the list of available products, as shown in Figure 2-20.

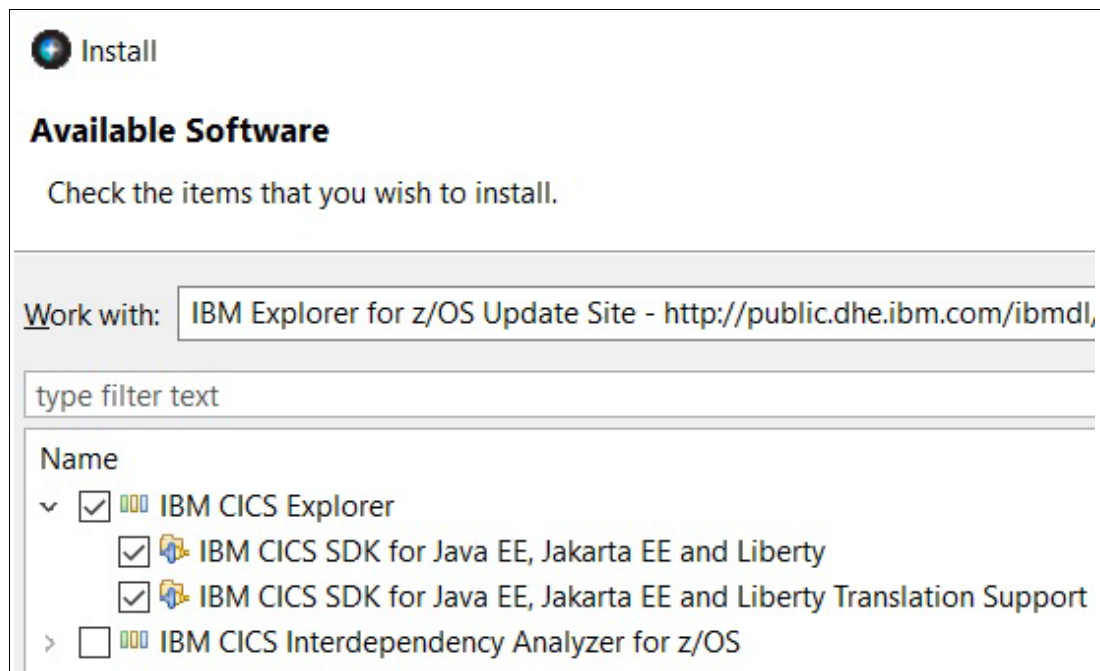


Figure 2-20 Installing CICS Explorer

6. We clicked the options to complete the installation and then, accepted the prompt to restart z/OS Explorer.

Downloading Liberty

Next, we must add IBM WebSphere Application Server Liberty.

To obtain the IBM Liberty product code, we went to this [web page](#).

We clicked the **Download** button, accepted the license agreement, and downloaded a file that is named `wlp-webProfile8-20.0.0.7.zip` (90 MB). We then unzipped this file to a parent directory that is named `C:\zProducts`.

Configuring Liberty runtime

To configure a Liberty runtime in z/OS Explorer, we selected **Window** → **Preferences**. In the displayed window, we expanded **Server** and select **Runtime Environments** and then, clicked **Add**. In the next window, we expanded IBM and selected **Liberty runtime**, as shown in Figure 2-21.

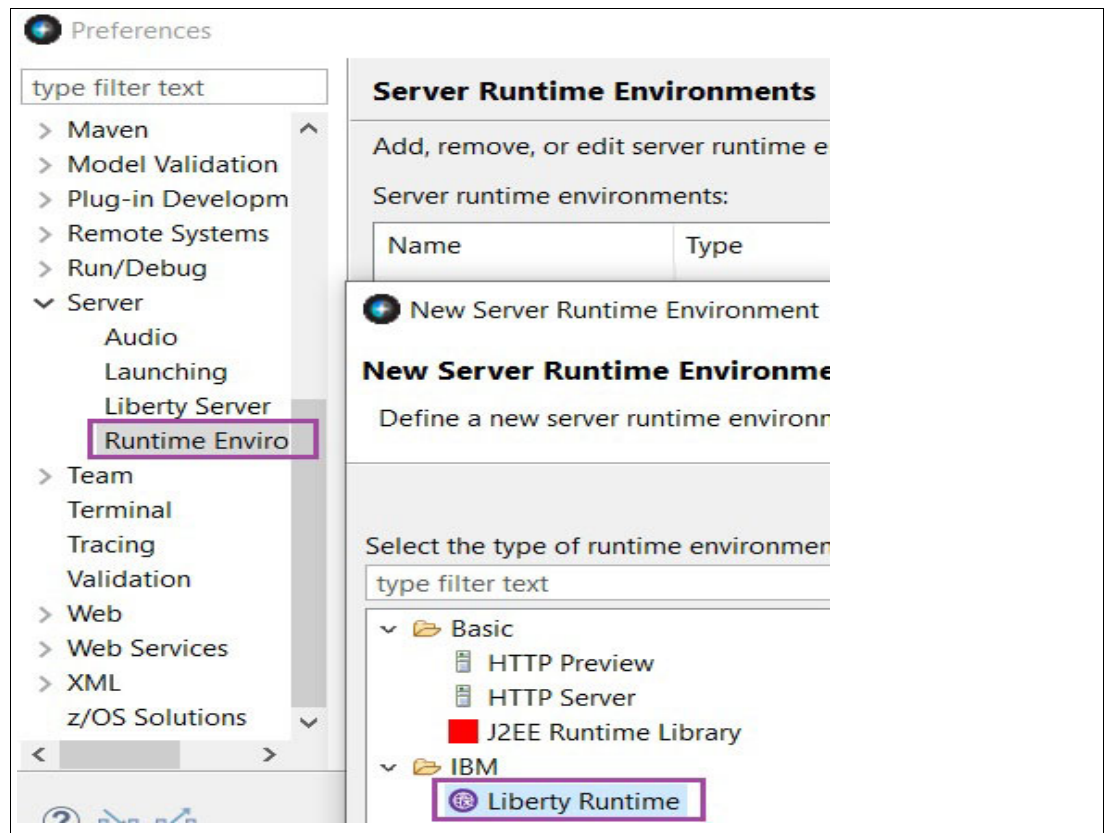


Figure 2-21 Setting up Liberty Runtime

7. We clicked **Next** and set the path to where we unzipped the Liberty product code, as shown in Figure 2-22.

The screenshot shows a Windows-style dialog box titled "New Server Runtime Environment". The main heading is "Liberty Runtime Environment" with a sub-instruction "Specify the runtime environment creation and JRE." and a Liberty logo. The "Name:" field contains "Liberty Runtime". Under "How do you want to create the runtime environment?", the "Choose an existing installation" radio button is selected. The "Path:" field shows "C:\zProducts\wlp" with a "Browse..." button. Below this, the "Install" radio button is also visible. The "JRE" section has two options: "Use a specific JRE:" (with "IBM Explorer for zOS" selected) and "Use default JRE (currently 'IBM Explorer for zOS')". A "Configure JRE" link is next to the default option. At the bottom are buttons for "?", "< Back", "Next >", "Finish" (highlighted with a blue border), and "Cancel".

Figure 2-22 Setting path to installed Liberty product code

2.10.7 Developing the ZKAFKA Java program

In this section, we describe how to develop the ZKAFKA Java program.

Creating a Dynamic Web Project

Complete the following steps:

1. Select **File** → **New** → **Dynamic Web Project**. Set the name as CicsToKafkaDemo and select the Liberty runtime, as shown in Figure 2-23.

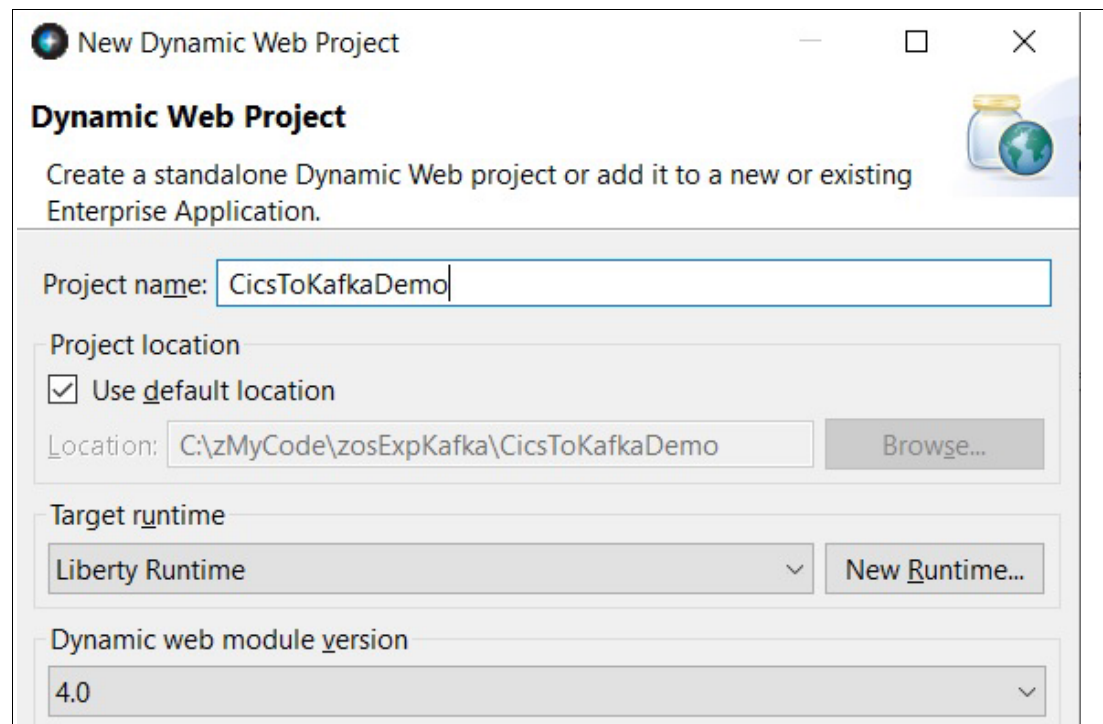


Figure 2-23 Creating Dynamic Web Project

2. Click **Finish** and the project appears in the Project Explorer view on the left. Select the **Java** perspective.

Importing Kafka JAR files

Complete the following steps:

1. Expand the project and right-click the **Lib** folder and select **Import**, as shown in Figure 2-24.

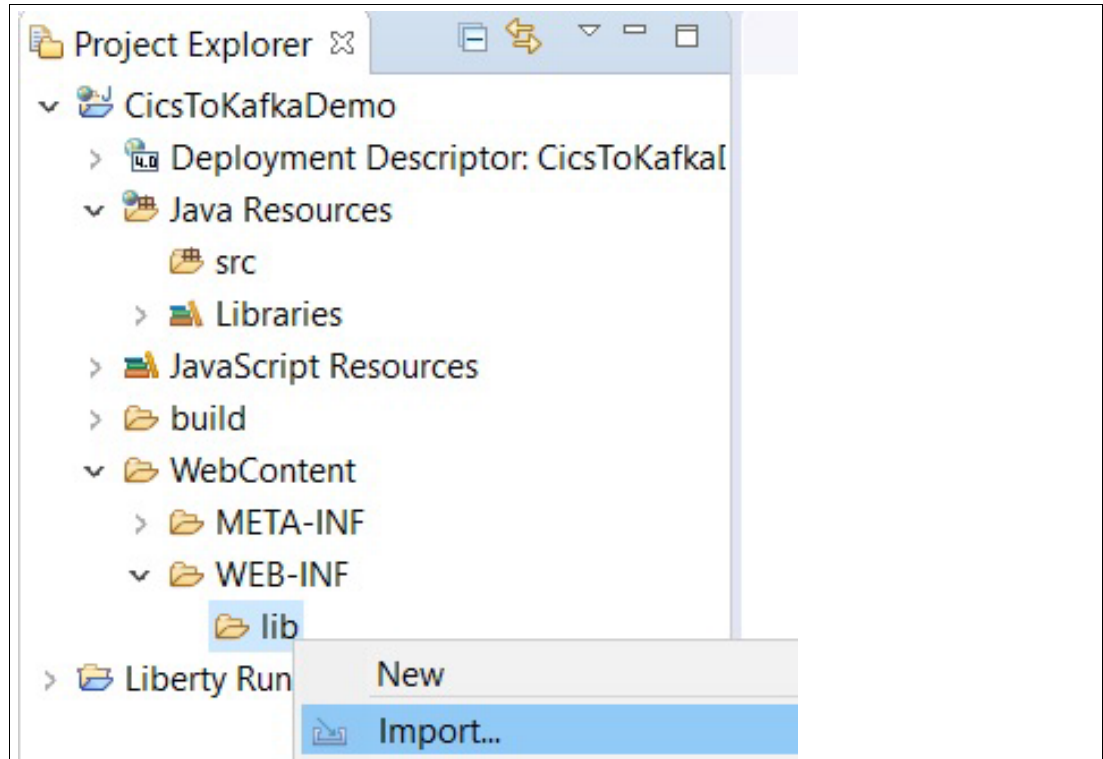


Figure 2-24 Start of import process

2. The Import window is displayed. Expand **General** and select **File System** and click **Next**.

3. In the next window, use the browse button to locate the Kafka .jar files that you downloaded and select **kafka-clients-2.5.0.jar** and **slf4j-api-1.7.30.jar**, as shown in Figure 2-25 on page 86.

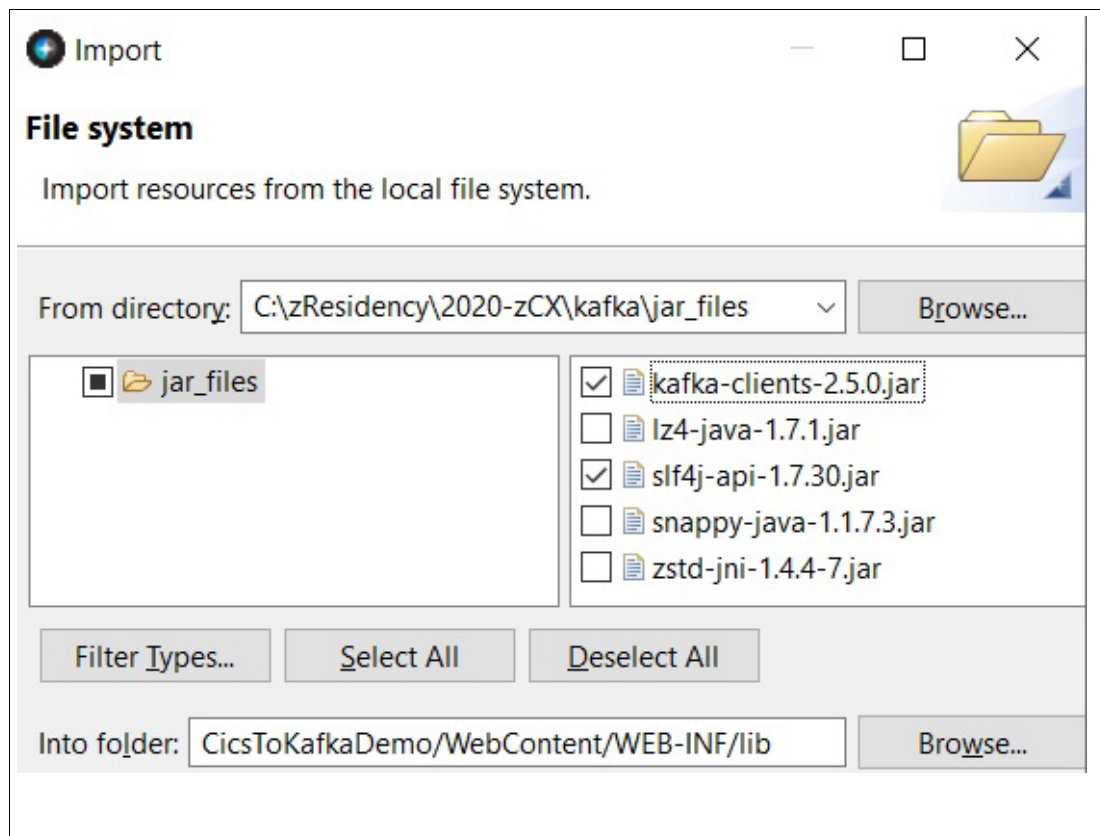


Figure 2-25 Importing Kafka JAR files

4. Click **Finish** and the .jar files then are shown under the Lib folder.

Updating Build Path

Complete the following steps:

1. Right-click the project and select **Build Path** → **Configure Build Path**, as shown in Figure 2-26.

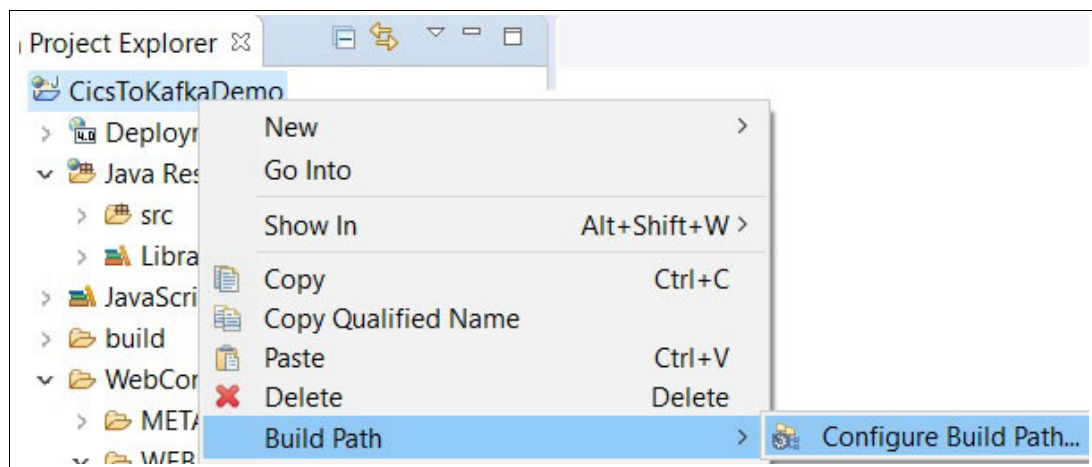


Figure 2-26 Configuring build path

2. Click **Next** and in the new window, select the version of CICS into which you are deploying. Click **Finish**.
3. Select the **Libraries** tab, then, **Add Library** then, select the CICS entry, as shown in Figure 2-27.

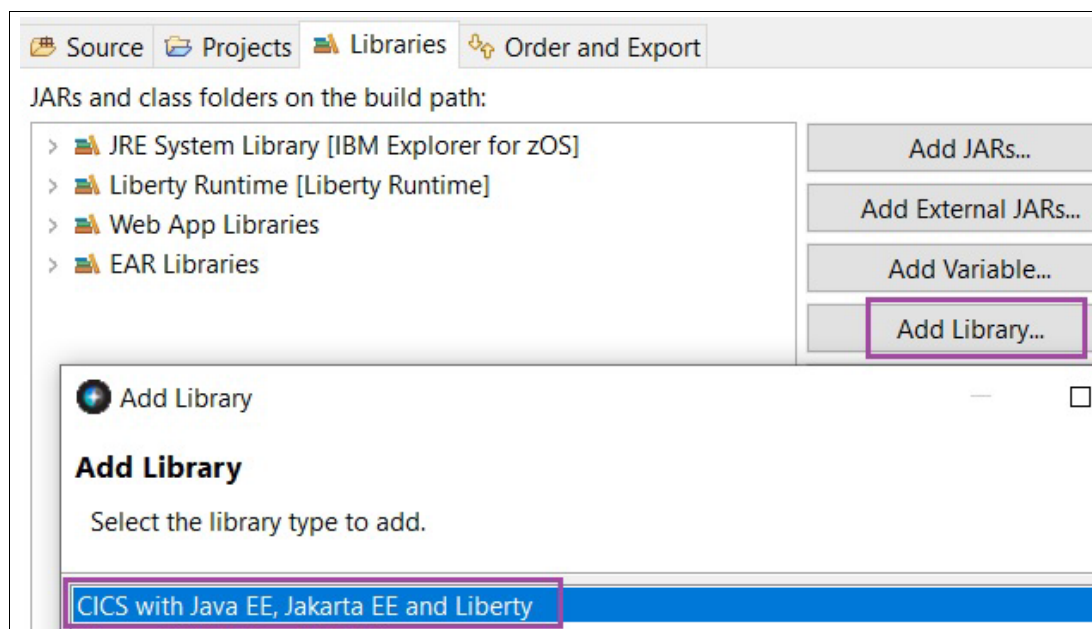


Figure 2-27 Adding CICS to the build path

4. Click **Apply and Close**.

Enabling annotation processing

The Java program includes the following line:

```
@CICSProgram("ZKAFKA")
```

This is a Java annotation. We need to ensure the project is configured to use this annotation. With annotation processing active in the project, more files are created in the war file.

When the CICS bundle that contains this program is installed into CICS, CICS deploys the Java program as a war file into the Liberty server. It then processes the files that are created by the annotation processing and automatically defines a CICS program definition for ZKAFKA.

You can verify that the required files were generated by expanding the created war file. Directories similar to the directories that are shown in Example 2-80 should be included.

Example 2-80 Other files created by annotation processing

```
C:\kafka\SendToKafka\WEB-INF\classes\com\ibm\cics\server\invocation>dir /S
```

Directory of C:\kafka\SendToKafka\WEB-INF\classes\com\ibm\cics\server\invocation

22/07/2020	10:39 AM	<DIR>	metadata
22/07/2020	10:39 AM	<DIR>	proxy
		0 File(s)	0 bytes

Directory of

```
C:\\kafka\SendToKafka\WEB-INF\classes\com\ibm\cics\server\invocation\metadata
```

22/07/2020	10:39 AM	490	com.ibm.kafkaDemo.SendEventToKakfa.xml
		1 File(s)	490 bytes

Directory of

```
C:\kafka\SendToKafka\WEB-INF\classes\com\ibm\cics\server\invocation\proxy
```

22/07/2020	10:39 AM	1,557	
			LinkBean_f5352257132d1712f559b6fd13d35b3f.class
		1 File(s)	1,557 bytes

Complete the following steps:

1. Right-click the project and select **Properties**.
2. In the next window, expand **Java Compiler** and select **Annotation Processing**.
3. Click **Enable Project Specific Settings**. The display should then look as shown in Figure 2-28.

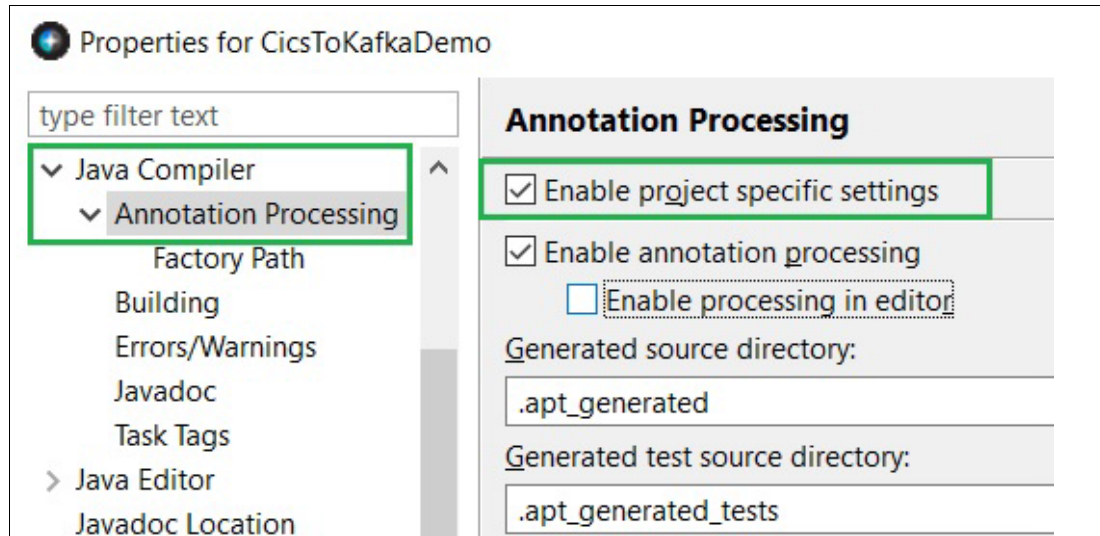


Figure 2-28 Enabling Annotation processing

4. Click **Apply and Close**.

Creating the Java class

Complete the following steps:

1. Right-click the **src** folder and select **New** → **Class**.
2. In the next the window, set the package name to `com.ibm.kafkaDemo` and the class to `CicsToKafka`, as shown in Figure 2-29 on page 90.

New Java Class

Java Class

Create a new Java class.

Source folder: CicsToKafkaDemo/src

Package: com.ibm.kafkaDemo

☐ Enclosing type:

Name: CicsToKafka

Modifiers: ☒ public ☐ package ☐ private ☐ protected

Figure 2-29 Create Java class

3. Click **Finish** and the Java class is created.
4. A window opens in which the generated source code is displayed. Delete that code and replace it with the contents of the supplied `CicsToKafka.java` program.
5. The code includes the lines that are shown in Example 2-81 that specify the TCP/IP address and port of the Kafka server to which you want to send. Change this information to suit your environment. If you plan to use a different Kafka topic, change the value for TOPIC as well.

Example 2-81 Target Kafka information

```
String KAFKA_BROKER_LIST = "129.40.23.77:19092";
// Specify name of Kafka topic to send to
String TOPIC = "fromCICSEVENT";
```

The program includes the following line:

```
@CICSProgram("ZKAFKA")
```

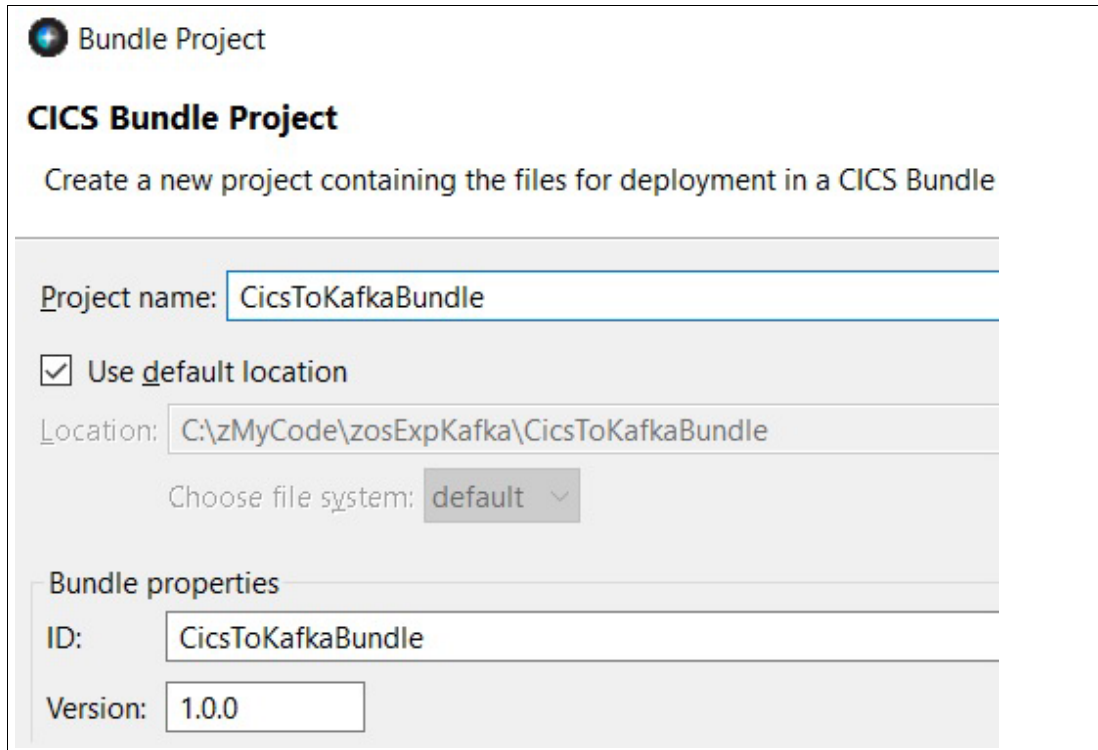
This line is the CICS program definition that is created when this program is deployed into CICS. Change it here, if required.

6. Save the changes.

2.10.8 Creating CICS bundle for Java program

A CICS bundle is used to deploy the Java program, as a war file, into the Liberty server in CICS. Complete the following steps:

1. Select **New** → **Other**. In the display window, expand **CICS Resources** and select **CICS Bundle Project**.
2. Set the project name to `CicsToKafkaBundle`, as shown in Figure 2-30.
3. Click **Finish**.

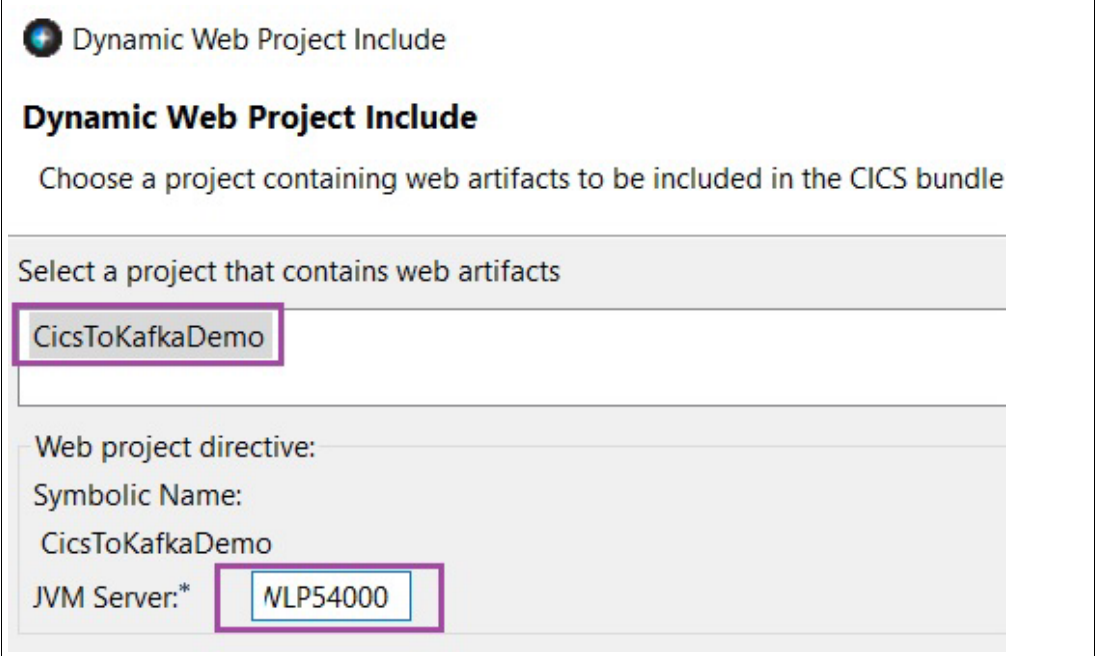


The screenshot shows the 'Bundle Project' wizard in the Eclipse IDE. The title bar says 'Bundle Project'. The main heading is 'CICS Bundle Project'. Below it, a subtitle reads 'Create a new project containing the files for deployment in a CICS Bundle'. The 'Project name' field is set to 'CicsToKafkaBundle'. The 'Use default location' checkbox is checked. The 'Location' field shows the path 'C:\zMyCode\zosExpKafka\CicsToKafkaBundle'. Below this, a 'Choose file system' dropdown menu is set to 'default'. The 'Bundle properties' section is expanded, showing 'ID' as 'CicsToKafkaBundle' and 'Version' as '1.0.0'.

Figure 2-30 Creating CICS bundle

4. Expand the created project. Then, expand the **META-INF** folder, and double click the **cics.xml** file.
5. In the displayed window under the Defined Resources heading, click **New** and select **Dynamic Web Project include**.

6. In the new window, select the **CicsToKafka** project, and set the name of the JVM Server. In our example, this name was WLP54000, as shown in Figure 2-31.



Dynamic Web Project Include

Dynamic Web Project Include

Choose a project containing web artifacts to be included in the CICS bundle

Select a project that contains web artifacts

CicsToKafkaDemo

Web project directive:

Symbolic Name:
CicsToKafkaDemo

JVM Server:* WLP54000

Figure 2-31 Adding Dynamic Web Project to CICS bundle

7. Click **Finish** and the project is shown under the Defined Resources heading.

Creating FTP connection to z/OS

You must export the projects to a UNIX System Services directory in z/OS.

You can manually process exporting to local file system on your PC and then manually FTPing to z/OS.

We show the use of a configured FTP connection to z/OS approach, which is simpler. Complete the following steps:

1. Open the z/OS Perspective.
2. Select the **Host Connections** tab and then, select **z/OS FTP**, and click **Add**, as shown in Figure 2-32.

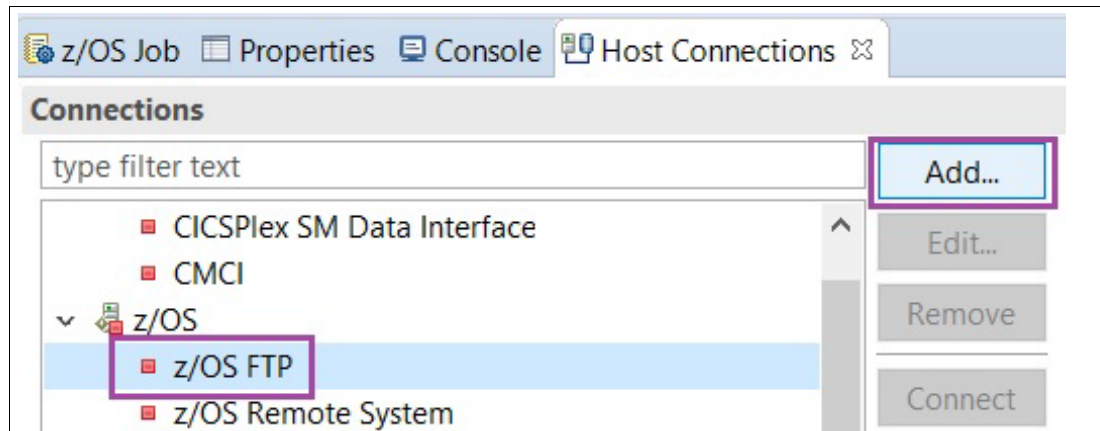


Figure 2-32 Start of process to create FTP connection to z/OS

3. In the window that opens, set the TCP/IP address of the z/OS LPAR and click **Save and Connect**.
4. In the next window, you are prompted to enter your sign on credentials, enter a user ID and password and click **OK**.

If the connection is successful, you should see a green dot displayed next to the entry that you created.

Exporting a CICS bundle to z/OS

Complete the following steps:

1. Right-click the **CicsToKafkaBundle** project and select **Export Bundle Project to z/OS UNIX File System**.
2. In the next window, select **Export to a specific location in the file system** and click **Next**.

3. In the next window, set the UNIX System Services directory name to export the bundle to (in our example, this directory is /global/zDevOps/cicscfg/cics_54/CD54000/bundles, as shown in Figure 2-33.

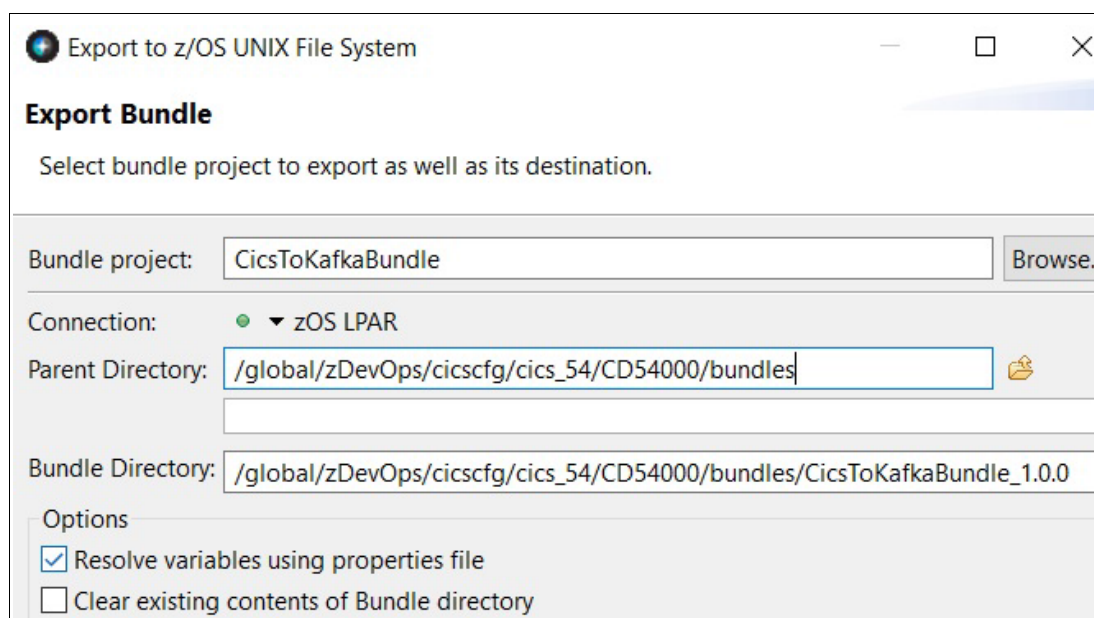


Figure 2-33 Exporting bundle to z/OS

4. Click **Finish** and messages that show the transfer to z/OS are displayed in the Console tab.

Creating a CICS CMCI connection

A CICS Bundle definition must be defined in CICS. This process can be done by way of CEDA transaction. We used the CICS Explorer to set up this definition and installation.

Complete the following steps:

1. Switch to the CICS SM explorer view and select the **Host Connections** tab. Then, select **CMCI** under **CICS System Management** and click **Add**, as shown in Figure 2-34.

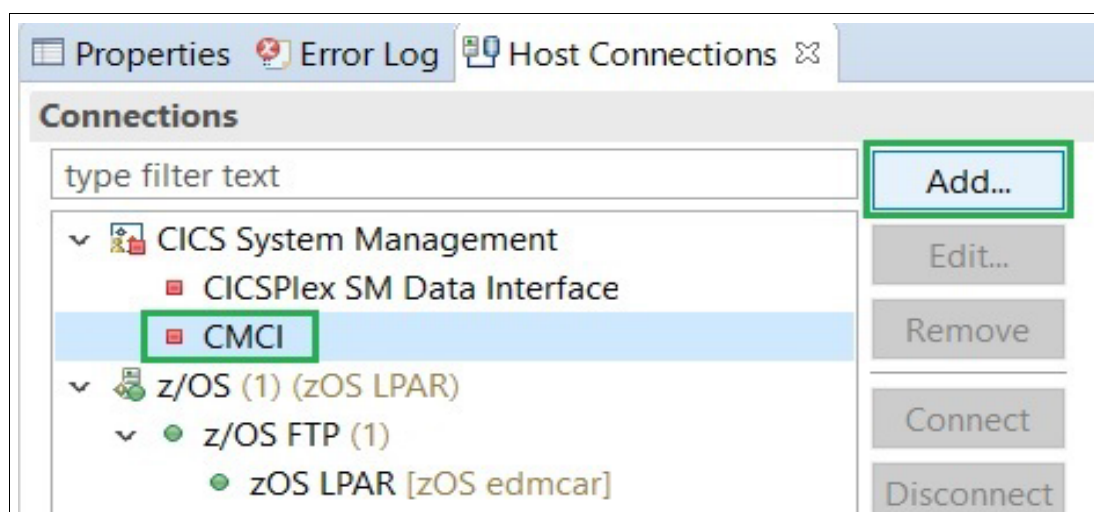


Figure 2-34 Defining CMCI connection to CICS

Figure 2-35 shows how we defined the required settings.

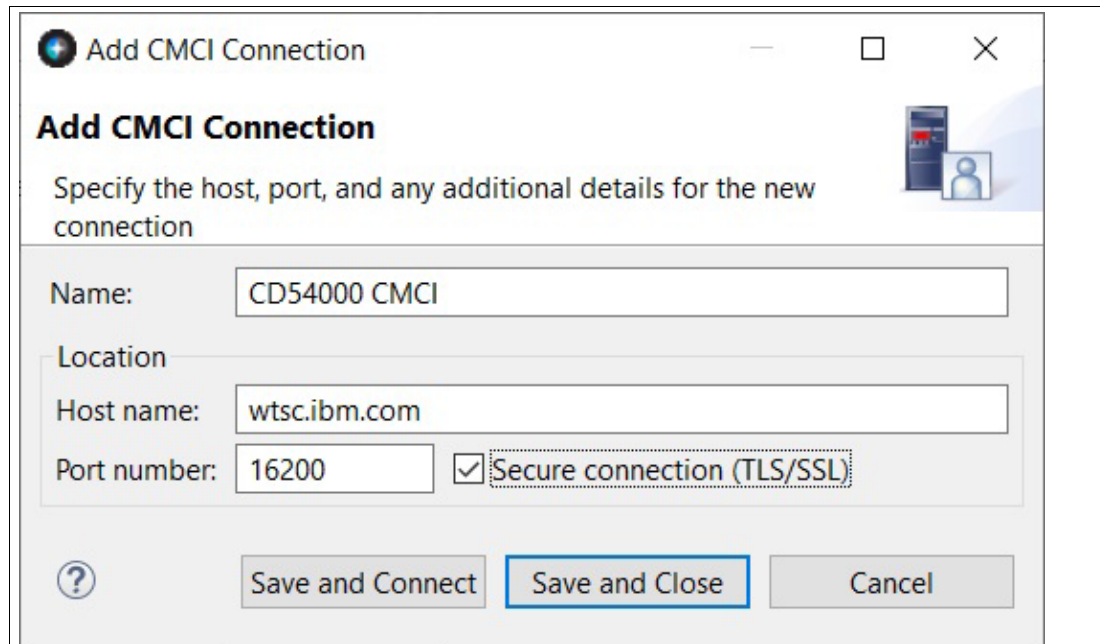


Figure 2-35 Settings for CMCI connection to CICS

2. Click **Save and Connect**.
3. Enter a user ID and password to connect. A successful connection shows a small green dot next to the defined connection.

Creating a CICS bundle definition

Complete the following steps:

1. Select the CICS region in the **CICSplex** tab then, select **Definitions** → **Bundle Definitions**, as shown in Figure 2-36.

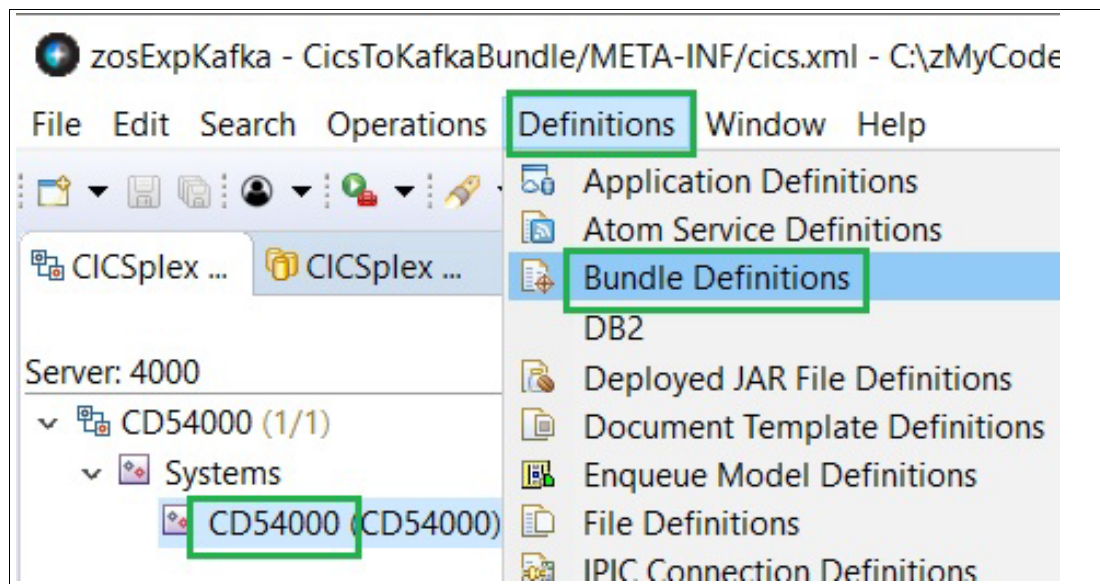
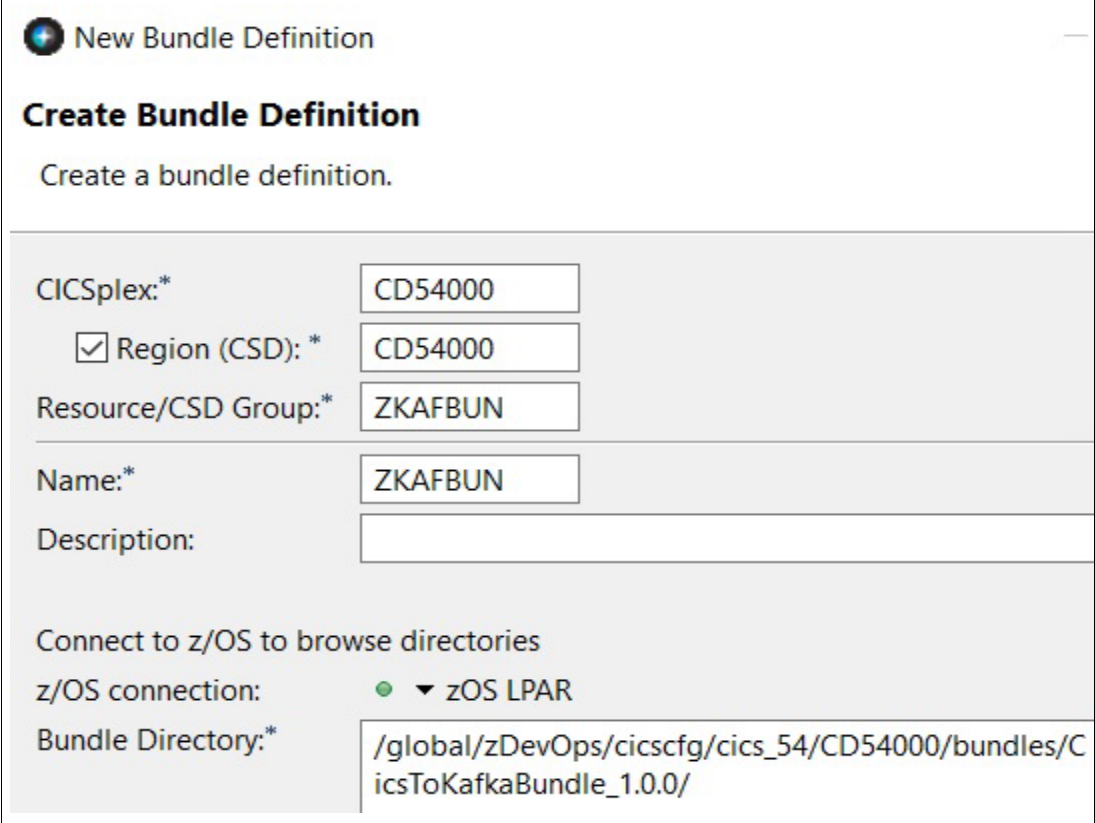


Figure 2-36 Selecting to define CICS Bundle definition

2. Enter a CSD group name and resource name, and specify the directory where the bundle is on z/OS. The values we used are shown in Figure 2-37.



New Bundle Definition

Create Bundle Definition

Create a bundle definition.

CICSplex:* CD54000


☒ Region (CSD): * CD54000

Resource/CSD Group:* ZKAFBUN

Name:* ZKAFBUN

Description:

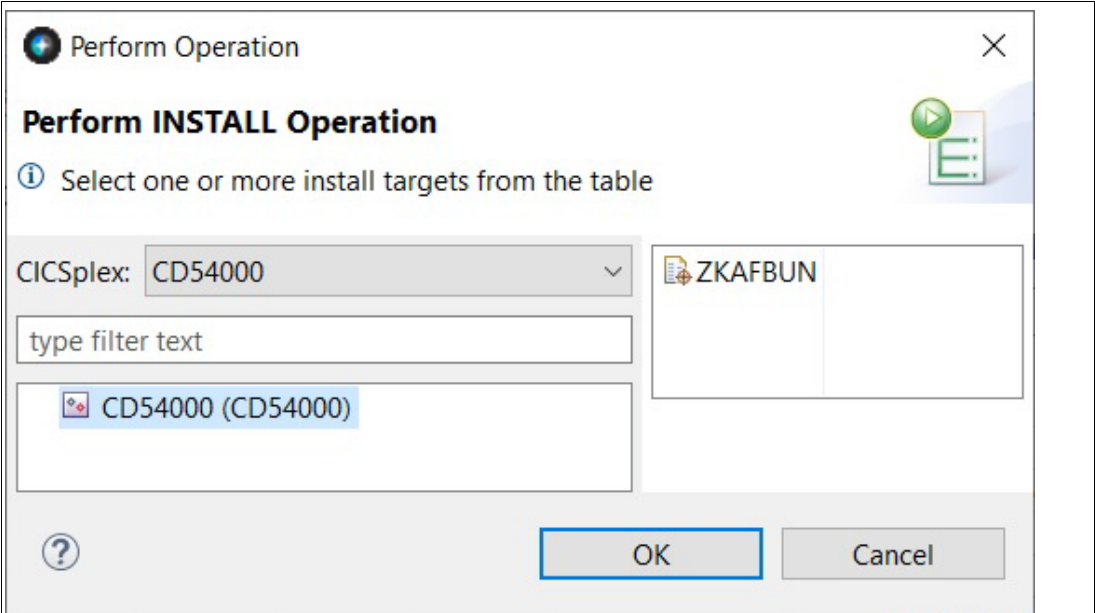
Connect to z/OS to browse directories

z/OS connection:  ▼ zOS LPAR

Bundle Directory:* /global/zDevOps/cicscfg/cics_54/CD54000/bundles/CicsToKafkaBundle_1.0.0/


Figure 2-37 Settings for CICS Bundle

3. Right-click the created definition and select **Install**. You see a display that is similar to the example that is shown in Figure 2-38.




Perform Operation


Perform INSTALL Operation

 Select one or more install targets from the table

CICSplex: CD54000 ▼

type filter text

 CD54000 (CD54000)

 ZKAFBUN


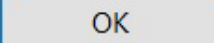
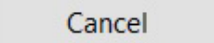
  

Figure 2-38 Installing the Bundle definition

4. Click **OK**.

The war file that is in the CICS bundle is then deployed into the CICS region. A CICS program definition that is named ZKAFKA also is created.

In the MSGUSER DD of the CICS region should be messages that is similar to the messages that are shown in Example 2-82.

Example 2-82 Messages showing Bundle install

```
DFHRD0128 I 07/22/2020 01:35:53 CD54000 EDMCAR CWWU INSTALL BUNDLE(ZKAFBUN)
???? ??? CWWU EDMCAR 07/22/20 01:35:53 INSTALL BUNDLE(ZKAFBUN) GROUP(ZKAFBUN)
DFHFC0961 07/22/2020 01:35:53 CD54000 Calculation of LSR pool 1 parameters
incomplete. Filename DFHDBFK has no DSNAMES.
DFHPG0101 07/22/2020 01:35:58 CD54000 CICSUSER ???? Resource definition for
ZKAFKA has been added.
DFHSJ1204 07/22/2020 01:35:58 CD54000 A linkable service has been registered for
program ZKAFKA in JVMSERVER WLP54000 with classname
com.ibm.kafkaDemo.CicsToKafka, method sendMsgToKafka.
```

The messages.log of the CICS Liberty server (which on our system was /u/edmcars/CD54000/WLP54000/wlp/usr/servers/defaultServer/logs) is where the messages that are shown in Example 2-83 output as a result of the installation of the bundle.

Example 2-83 Liberty messages

```
CNTR4000I: The CicsToKafkaDemo.war EJB module in the CicsToKafkaDemo application
is starting.
CNTR0167I: The server is binding the
com.ibm.cics.server.invocation.proxy.LinkBean_alf8c1cca768f8091667b30ca62d57d9
interface of the LinkBean_alf8c1cca768f8091667b30ca62d57d9 enterprise bean in the
CicsToKafkaDemo.war module of the CicsToKafkaDemo application. The binding
location is:
java:global/CicsToKafkaDemo/LinkBean_alf8c1cca768f8091667b30ca62d57d9!com.ibm.cics
.server.invocation.proxy.LinkBean_alf8c1cca768f8091667b30ca62d57d9
CNTR4001I: The CicsToKafkaDemo.war EJB module in the CicsToKafkaDemo application
has started successfully.
SRVE0169I: Loading Web Module: CicsToKafkaDemo.
SRVE0250I: Web Module CicsToKafkaDemo has been bound to default_host.
CWWKT0016I: Web application available (default_host):
http://sc74cn02.pbm.ihost.com:19450/CicsToKafkaDemo/
SESN0176I: A new session context will be created for application key
default_host/CicsToKafkaDemo
SESN0172I: The session manager is using the Java default SecureRandom
implementation for session ID generation.
CWWKZ0001I: Application CicsToKafkaDemo started in 1.717 seconds.
```

Testing Java Program

Although the CICS event is not yet set up, we can test whether the program is callable by using CECI transaction in CICS. In CICS, enter the following command:

```
CECI LINK PROGRAM(ZKAFKA)
```

You should get response code of zero. The /u/edmcars/CD54000/WLP54000/CURRENT.STDOUT file included the following message:

```
2020/07/22 05:47:31>>>There is no Current Channel
```

Because the Java program did not yet pass a CICS channel, this error message is output, but it does show that the Java program can be LINKed to in a CICS region.

2.10.9 Defining a CICS event

In this section, we show how to configure a CICS event to collect data when the VSAM file EXMPCAT is updated.

Defining CICS ZOEV transaction

The CICS event we create is configured to start a transaction that is named ZOEV.

In CICS, define a program definition to start the program that is named ZKAFKA and install it.

Defining a new bundle project

Select the CICS SM perspective and then, select **New** → **CICS Bundle Project**, as described in 2.10.8, “Creating CICS bundle for Java program” on page 91.

In the new window, set the project name to `CatalogEvents` and click **Finish**.

Adding sample Catalog REWRITE event

CICS supplies a sample event that acts on the VSAM REWRITE that is done by the CICS Catalog sample application. We can use that sample event after a few changes are made.

Complete the following steps:

1. Right-click the created project and select **New** → **Other**.
2. In the next window, select **Examples** → **CICS Examples** → **Event Binding** → **Catalog Manager REWRITE Event**, as shown in Figure 2-39.

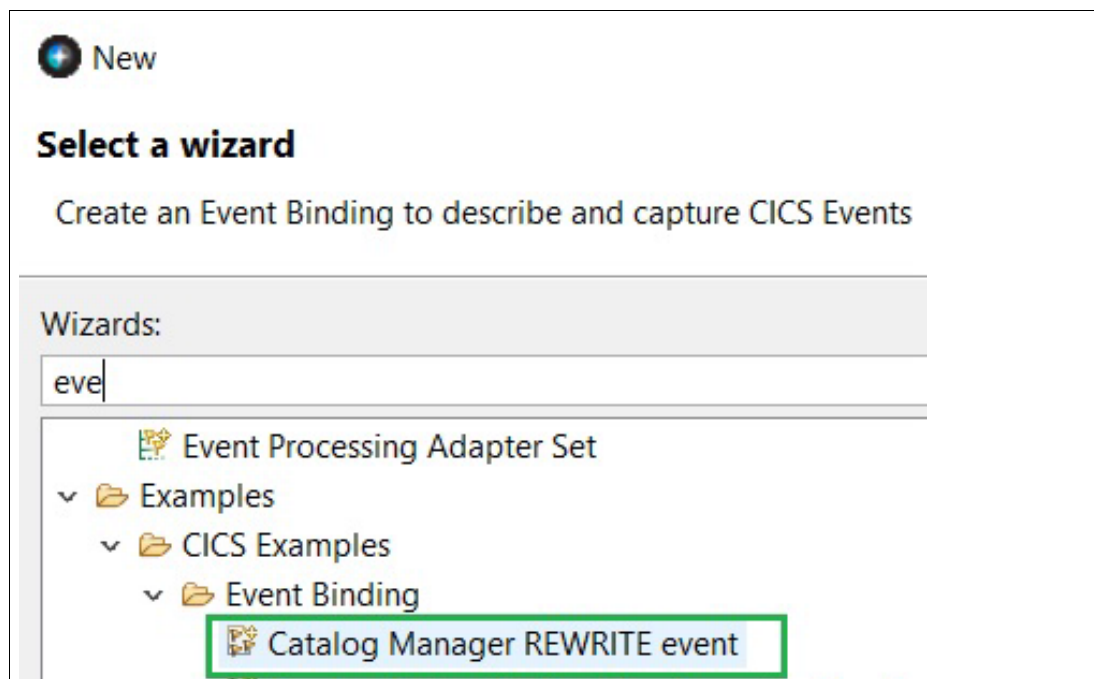


Figure 2-39 Using the sample Catalog event

The sample CICS Event is imported into the project.

3. In the next window, set the file name. In our example, we set it to `CatalogUpdate`, as shown in Figure 2-40.

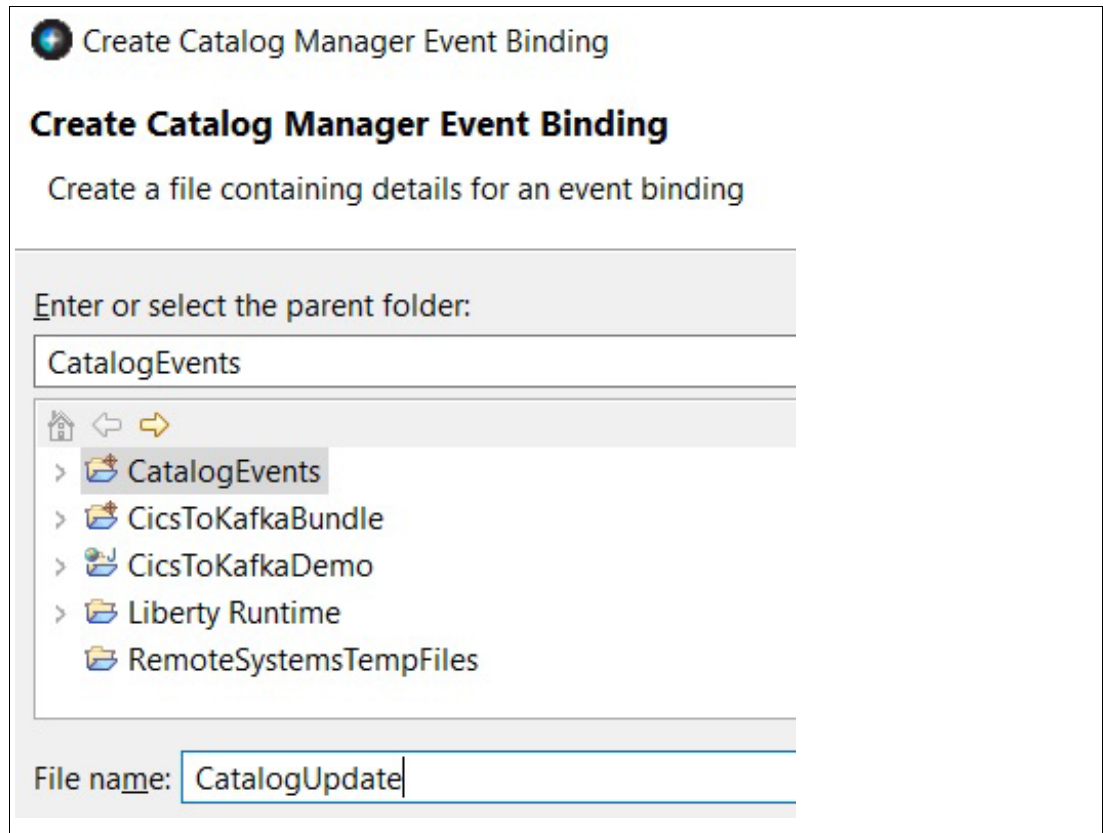


Figure 2-40 Adding sample event to project

4. Click **Finish** and a file that is named `CatalogUpdate.evbind` is created in the project.

Changing event filtering

The event is configured to generate only an event when the update is done, and the number of items still in stock is less than 24. For this example, we want to trigger an event whenever an update is done.

Complete the following steps:

1. Double click the **CatalogUpdate.evbin** file to open it and expand **Catalog_stock_status_check**. Double click **Check_stock_status_on_rewrite**.
2. Click the **Filtering** tab, select the first entry under the **Application Data** heading and click **Edit**, as shown in Figure 2-41.

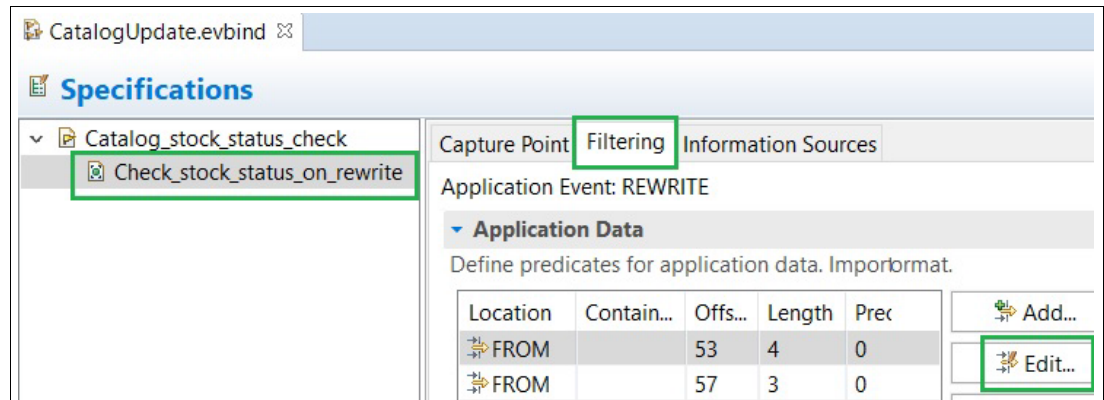


Figure 2-41 Selecting the filter to edit

3. Change the value to 2000, as shown in Figure 2-42. This higher value should result in an event being generated whenever the VSAM file is updated.

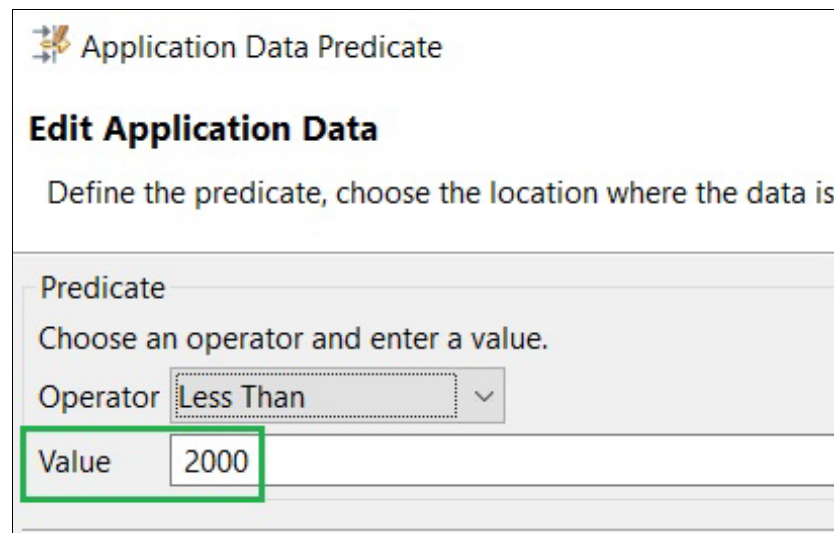


Figure 2-42 Changing filter test to 2000

4. Save the change.

Changing an event to start ZOEV transaction

Complete the following steps:

1. Click the **Information Sources** tab and in the lower right of the window, select **Next Adapter**.
2. The sample event writes the event to a temporary storage queue, but we want it to start a CICS transaction that in turn starts our ZKAFKA Java program. In the **Adapter** tab, select **Transaction Start** from the drop-down list and specify **ZOEV** as the transaction to be started by the event, as shown in Figure 2-43.

The screenshot shows the 'Adapter' configuration window. The 'Resource' section has three radio buttons: 'Use a predefined EPADAPTER resource', 'Use a predefined EPADAPTERSET resource', and 'Use an adapter defined here' (which is selected). Below this is a button 'Export Event Specifications...'. The 'Adapter' section has a dropdown menu with 'Transaction Start' selected. Below the dropdown is a text description: 'Emits events to a named CICS transaction transaction. You can use an existing trans'. At the bottom, the 'Transaction ID' field is set to 'ZOEV'.

Figure 2-43 Changing the Adapter to start a CICS transaction

3. Save this change.

Exporting bundle to UNIX System Services directory on z/OS

Use the process that is described in “Exporting a CICS bundle to z/OS” on page 93 to export the bundle definition to a directory in UNIX System Services on the z/OS LPAR.

Defining CICS bundle definition for event

Use the process that is described in “Creating a CICS bundle definition” on page 95 to create a bundle definition and then install it into the CICS region.



IBM App Connect Enterprise

IBM App Connect Enterprise for zCX on z/OS can provide the flexibility to integrate z/OS subsystems close to the source and targets, including IIMS, Db2, CICS and IBM MQ.

In this chapter, we demonstrate this flexibility with a use case that deploys an application to ACE that integrates with CICS.

This chapter includes the following topics:

- ▶ 3.1, “Technical and architectural concepts of ACE” on page 104
- ▶ 3.2, “Installing IBM App Connect Enterprise” on page 109
- ▶ 3.3, “Configuration details” on page 118
- ▶ 3.4, “Deploying an application to ACE to integrate with CICS” on page 119

3.1 Technical and architectural concepts of ACE

Introduced in 1999, IBM MQ Series integrator was rebranded as WebSphere Business Integration Message Broker and then WebSphere Message Broker. In 2013, IBM Integration Bus (IIB) v9 enabled users to import and convert integration assets from WebSphere Enterprise Service Bus, which converged these capabilities into IIB.

In 2018, IBM released App Connect Enterprise (ACE) which combined the existing, industry-trusted technologies of IBM Integration Bus (IIB) with IBM App Connect capabilities (professional software and Cloud Connectors by way of IBM Cloud® managed service). The run time of IIB was also updated in this release to make it cloud native so the term “Bus” in the name did not accurately reflect the possibilities that this new architecture opened up.

IBM App Connect Enterprise redesigned the core IBM Integration Bus technology to make it amenable for deployment in container-based architectures while also continuing support for older design paradigms, such as the Enterprise Service Bus pattern. This redesign makes IBM App Connect Enterprise v11 an excellent choice for production systems for users who want to embrace the benefits of containers as part of an Agile Integration Architecture, and users who want to make minimal changes to their IBM Integration Bus architecture yet benefit from the new features and product lifespan of v11.

Figure 3-1 shows the evolution of IBM App Connect Enterprise (ACE).

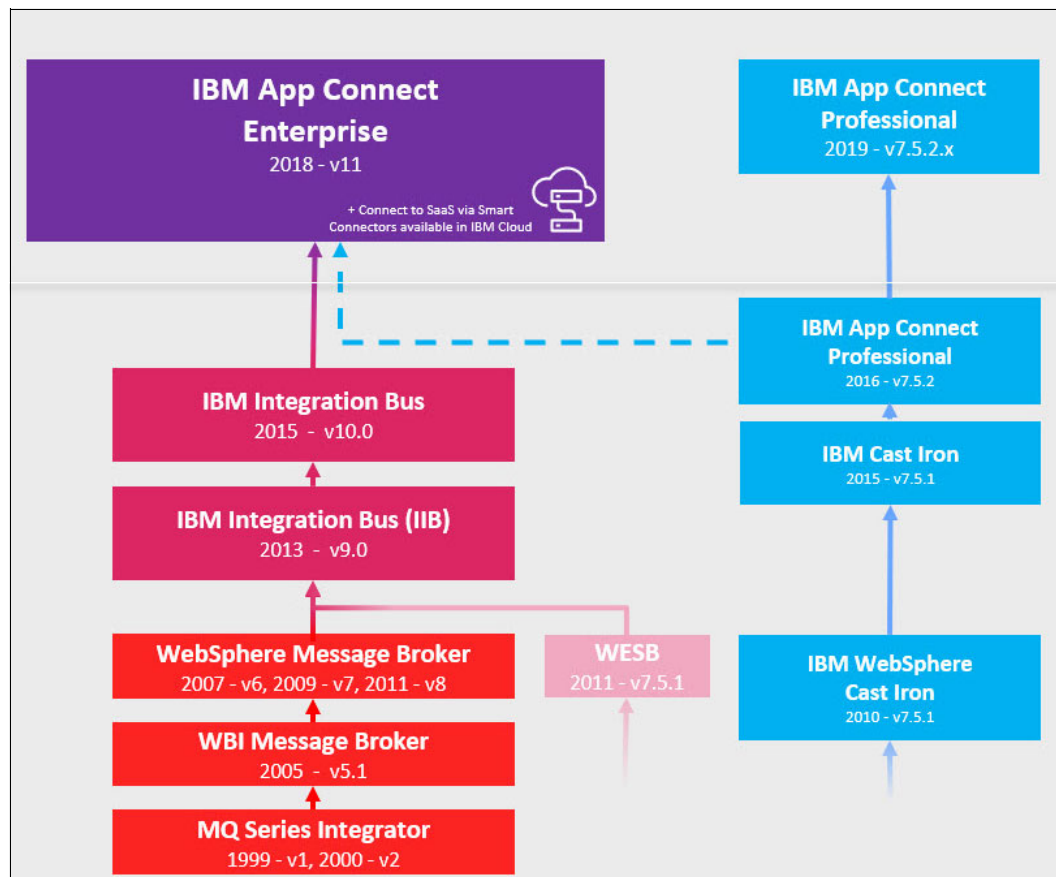


Figure 3-1 Evolution® of IBM App Connect Enterprise

Before we discuss IBM App Connect Enterprise, we discuss what we mean by the term *integration*. There are three concepts that are involved in connecting applications:

- ▶ **Endpoint:** The different communication protocols.
- ▶ **Inputs:** The data that is exchanged between endpoints in different formats.
- ▶ **Integration:** The mediation patterns for interoperability between endpoints: message transformation, enrichment, audit, aggregation, and scaling.

Applications must communicate with each other over a communications protocol. Typical protocols in use today include TCP/IP, and higher-level protocols, such as IBM MQ, FTP, SMTP, and HTTP.

Applications exchange data over the communications protocol, typically in discrete structures known as *messages*. The format of these messages can be defined from JSON, C structures or COBOL copybooks (for example), or use a standard format, such as XML.

To connect applications so that their protocols and message formats interoperate, mediation patterns must be applied to one or both systems that you are trying to connect. These mediation patterns can be relatively straightforward; for example, routing messages from one place to another, or the transformation of one message format into another to relatively complex patterns, such as aggregating multiple outputs from an application into a single message for a target system.

IBM App Connect Enterprise enables “universal connectivity” by integrating protocols, message formats, and mediation patterns.

ACE provides the ability to be an endpoint and to connect to other endpoints. It can do this over various protocols and by using various message formats, sometimes with more than one in use at each time.

ACE also supports a wide range of mediation patterns, which helps to support the use of the various message formats and protocols in many ways. Figure 3-2 shows the positioning of ACE in an enterprise.

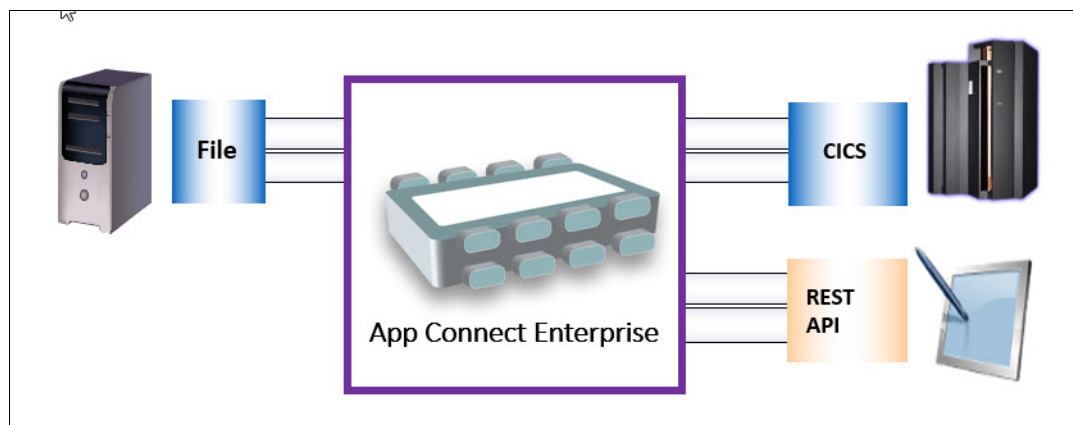


Figure 3-2 ACE in an enterprise

3.1.1 Key concepts of ACE

Figure 3-3 shows some of the key concepts of ACE.

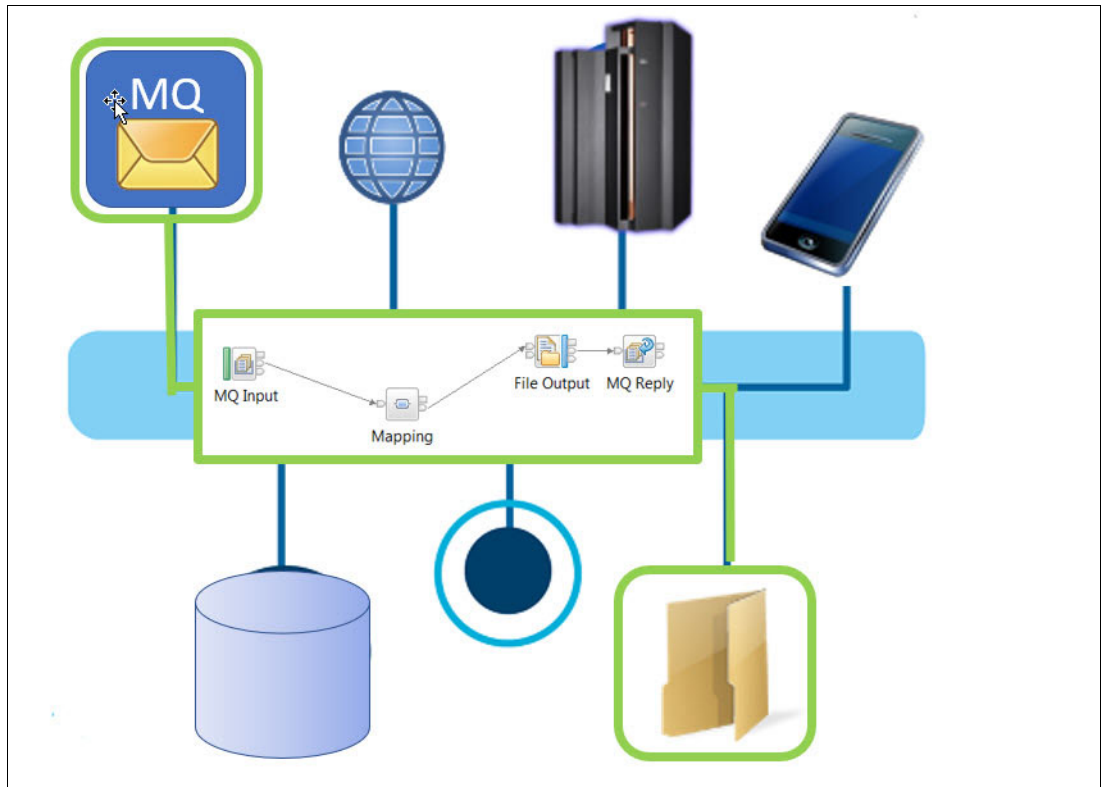


Figure 3-3 Key concepts of ACE

The image in the middle of Figure 3-3 represents the IBM App Connect Enterprise run time. It can connect multiple different applications and systems. Many types of applications can be connected, including the following examples:

- ▶ IBM MQ applications
- ▶ Mainframe applications (such as CICS or IMS) or Enterprise Information Systems, such as SAP
- ▶ Databases
- ▶ Files
- ▶ REST APIs
- ▶ Web services

In ACE, we can define specific integrations between one system and one or more other systems. For example, an IBM MQ application can be integrated with CICS and writing to a file-based application.

In ACE, the logic of this integration is described by a message flow. A message flow is way to write a program where reusable blocks called *nodes* are dragged to the canvas and wired together with some transformation code and integration logic. Message flows are general purpose, reusable integration applications.

The message flow is a key concept in IBM App Connect Enterprise/IBM Integration Bus.

3.1.2 Runtime components of ACE

In this section, we provide an overview of the ACE product architecture. Figure 3-4 shows the runtime components of ACE.

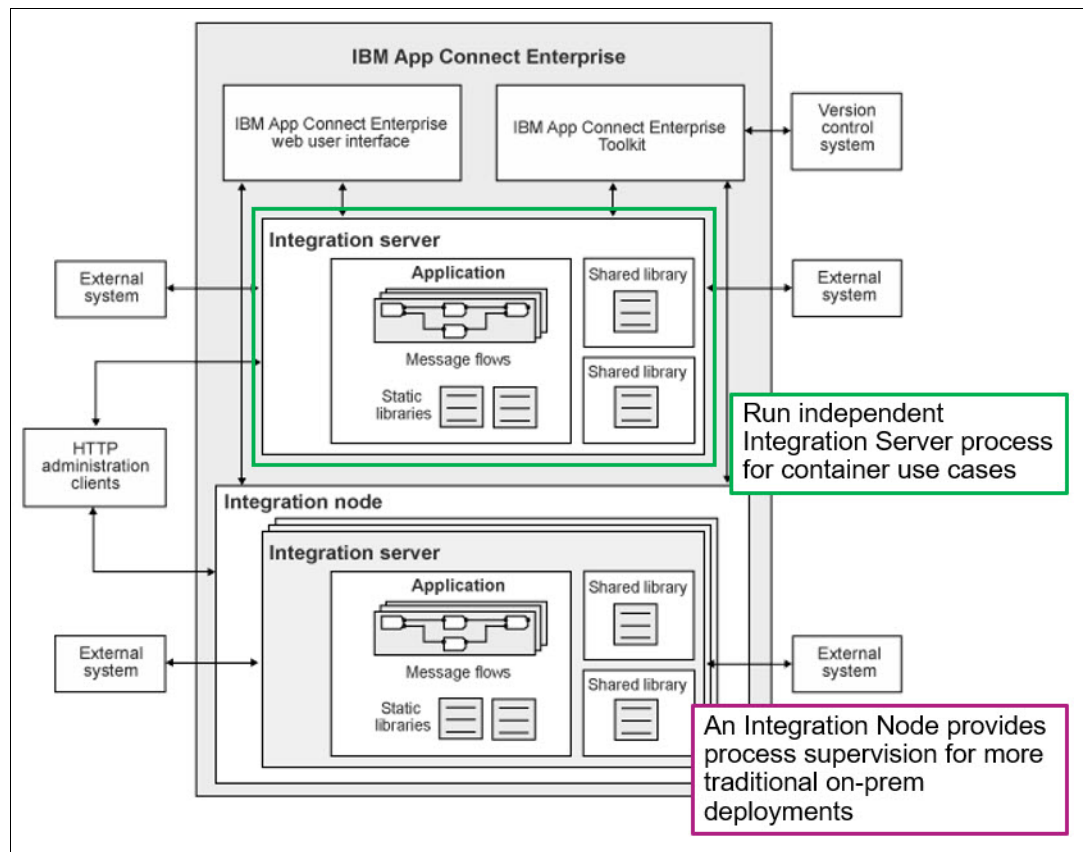


Figure 3-4 Runtime components of ACE

ACE includes the following main components:

- Integration Toolkit

The Integration Toolkit is the development environment. Based on the Eclipse platform, all the objects that are required to perform application integration by using ACE are developed, deployed, and tested here. It provides standard ways to build integration applications and perform version control.

- Integration node and integration server

The integration node (or broker) is the container that hosts integration servers (or execution groups). Each integration server is an operating system process that contains a pool of threads that are responsible for running the message flows that are deployed to it.

Message flows are deployed to integration servers in applications that might contain reusable sets of resources. The integration servers directly interact with the endpoints that are being integrated.

- Web UI

The web user interface (Web UI) provides administration capability, including monitoring of deployed objects and the ability to start, stop, delete, deploy, and manage workload.

3.1.3 ACE run time in zCX

In IBM App Connect Enterprise integration servers can be deployed in one of two ways: one or more integration servers under the management of an integration node or as independent integration servers.

In container architectures, such as zCX, you run the ACE run time as independent integration servers. They are started directly through an external framework within a Docker container. The independent integration runtimes are used for container-based and micro-services aligned architectures.

Figure 3-5 shows an overview of the ACE runtime integration server in zCX.

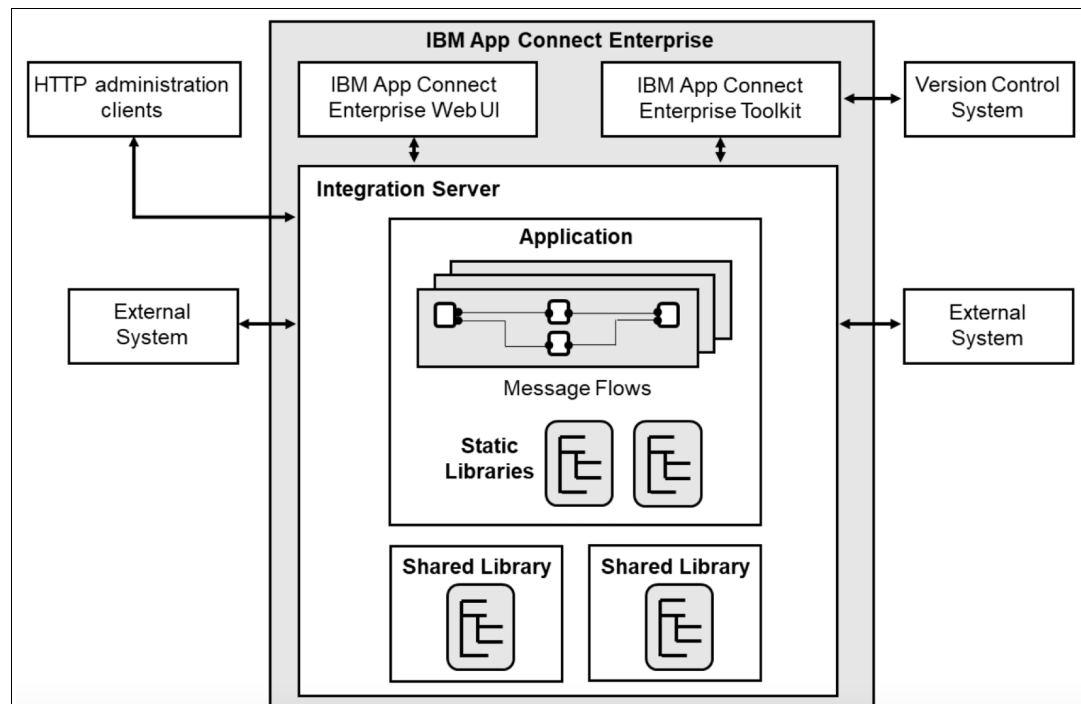


Figure 3-5 Integration Server in zCX

This lightweight, cloud-native runtime (integration server) can be used in the following ways:

- ▶ As part of a DevOps or agile approach: The lightweight container spins up in seconds.
- ▶ In a microservices model, which is managed and deployed by the microservice teams close to the microservice.
- ▶ Across multiple clouds: Private, public, or as a fully managed cloud service.

The independent integration server also includes the following key features:

- ▶ Deploy an application to an Integration Server from Toolkit, REST Commands, and Web UI.
- ▶ Display one of more local or remote independent integration servers, which can be deployed to.
- ▶ Toolkit communications with servers by way of an administrative REST API. Bar files are deployed to servers by using this REST API.
- ▶ View and administer independent integration servers in the Web UI.

3.1.4 Reasons to run ACE on zCX

ACE is run on zCX for the following three main reasons:

- ▶ Server consolidation

If you are running ACE integration servers off platform, you can now move them to z/OS and manage them along with applications and data they serve, such as CICS or IMS applications.

It also means that the z/OS qualities of service, such as scalability, availability, integrated disaster recovery, backup, WLM, and security are available to the integration servers.

- ▶ Save money

If you are running IIB v10 on z/OS and have any particularly MIPS heavy message flows, you might consider moving those flows into an environment where most of the processing can be offloaded to zIIP processors to reduce costs.

- ▶ Skills

If deep IBM Integration Bus experience lies in z/OS system programmers, run integration servers where knowledge and experience lies.

IBM App Connect Enterprise on zCX enables z/OS customers to run and manage ACE in zCX by using JCL or z/OS console commands. For more information, see [IBM Knowledge Center](#).

3.2 Installing IBM App Connect Enterprise

As of this writing, no prebuilt image is available to install IBM ACE in a zCX image. However, it is possible to install ACE by using a step-by-step approach.

In this section, we explain how to install and get ACE up and running. A key point to remember is that you can run the v11 (Fix Pack 8 onwards) ACE integration server on zCX.

3.2.1 Obtaining the ACE installation binaries

Complete the following steps to obtain the ACE installation binaries:

1. Log on to the zCX instance where the ACE Image should be built. After logging on, the default path is /home/admin. Run the following command:

```
mkdir redbooks && cd redbooks
```
2. Download the ACE binaries from IBM Support into the newly created redbooks folder that is used when building the image. In this publication, we used V11.0.0 with fixpack FP0010 with a release date of the 28 May 2020.

The most recent fixes can be found at this IBM Support [web page](#).

At this web page, select the **Fix Central Download** image for the Linux on zSeries platform.

Important Note: You must have an IBM account to download ACE from the IBM Support server and a valid licence coupled to the IBM ID that is used to download the image.

To download the image directly into the zCX instance, select **FTPS/SFTP** as the download option, as shown in Figure 3-6 on page 110.

Download options

WebSphere, IBM App Connect Enterprise (All releases, All platforms)

 [Subscribe to support notifications](#)

Select download options

Select the download method to be used to download fixes.

☐ Download using Download Director
(requires Java)

 [What is this?](#)

☒ Download using bulk FTPS/SFTP

 [What is this?](#)

☐ Download using your browser (HTTPS)

CAUTION: Do not assume that Fix Central will show you all the prerequi

Figure 3-6 Available download options

Use the provided user ID and password to download the file, as shown in Example 3-1.

Example 3-1 Download ACE image from IBM Support

```
admin@sc74cn11:~/redbooks/$ sftp <your_user>@delivery04-bld.dhe.ibm.com
The authenticity of host 'delivery04-bld.dhe.ibm.com (170.225.15.104)' can't be
established.
RSA key fingerprint is SHA256:QRJp0HdTFuPmP2NLOQHTpB+IrDSNrque7RadzKcFyFc.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added 'delivery04-bld.dhe.ibm.com,170.225.15.104' (RSA) to
the list of known hosts.
ASMBggPF@delivery04-bld.dhe.ibm.com's password:
Connected to delivery04-bld.dhe.ibm.com.
sftp> mget *
Fetching /11.0.0-ACE-LINUXZ64-FP0010.tar.gz to 11.0.0-ACE-LINUXZ64-FP0010.tar.gz
/11.0.0-ACE-LINUXZ64-FP0010.tar.gz
100% 690MB 10.1MB/s 01:08
sftp>
exit
```

3. Verify that the binary file was downloaded by running the command that is shown in Example 3-2.

Example 3-2 List downloaded file

```
admin@sc74cn11:~/redbooks$ ls -la
total 706612
```



```
drwxrwxr-x 1 admin admin          66 Sep 10 18:44 .
drwxr-xr-x 1 admin admin          222 Sep 10 18:44 ..
-rw-r----- 1 admin admin 723570314 Sep 10 18:41 11.0.0-ACE-LINUXZ64-FP0010.tar.gz
admin@sc74cn11:~/redbooks$
```

3.2.2 Obtaining the IBM App Connect Enterprise Docker container build source

In this section, we provide two ways of obtaining the IBM App Connect Enterprise Docker container build source:

- ▶ “Pulling ACE Docker container build source from GitHub”
- ▶ “Building the IBM App Connect Enterprise Docker container build source” on page 112

Pulling ACE Docker container build source from GitHub

Complete the following steps to pull the ACE Docker container build source from GitHub:

1. Run the command that is shown in Example 3-3 to pull data from GitHub.

Example 3-3 Pull ACE Docker container build source from GitHub

```
admin@sc74cn11:~/redbooks$ curl -Lk https://github.com/ot4i/ace-docker/tarball/master | tar xz
% Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
           % Dload  % Upload   Total   Spent    Left   Speed
100  130    100  130    0    0  1101      0 --:--:-- --:--:-- --:--:--  1101
100 313k    0 313k    0    0 942k      0 --:--:-- --:--:-- --:--:-- 942k
```

2. Check the name of the directory that was created by running the **ls** command, as shown in Example 3-4.

Example 3-4 Getting the name of ACE data

```
admin@sc74cn11:~/redbooks$ ls
ot4i-ace-docker-6c430bc
```

3. Rename the directory (in our case, `ot4i-ace-docker-6c430bc`) to `ace-docker` by running the **mv** command and list the contents of the `ace-docker` directory, as shown in Example 3-5.

Example 3-5 Renaming the ACE data directory

```
admin@sc74cn11:~/redbooks$ mv ot4i-ace-docker-6c430bc/ ace-docker/
admin@sc74cn11:~/redbooks$ cd ace-docker/
admin@sc74cn11:~/redbooks/ace-docker$ ls
CHANGELOG.md          ace_config_agent.sh      ace_config_odbcini.sh
ace_config_truststore.sh ace_license_check.sh      build-rhel.sh
go.sum                ubi
Jenkinsfile           ace_configBars.sh        ace_config_policy.sh
ace_config_webusers.sh ace_mqsiCommand.sh        cmd
internal
LICENSE               ace_config_extensions.sh ace_config_serverconf.sh
ace_discover_port_overrides.sh app_connect_light_256x256.png deps
jenkins-build-scripts
README.md             ace_config_keystore.sh    ace_config_setdbparms.sh
ace_env.sh            appconnect_enterprise_logo.svg experimental
licenses
```

```
ace_compileBars.sh  ace_config_logging.sh  ace_config_ssl.sh
ace_integration_server.sh  asoc  go.mod
sample
```

Building the IBM App Connect Enterprise Docker container build source

If you do not want to download the ACE Docker files and scripts from GitHub by using the process that is described in “Pulling ACE Docker container build source from GitHub” on page 111, you can use a Git clone and pull the ACE Docker container files and scripts.

Complete the following steps to build an intermediate image to download more data from the ace-docker Git repository that is available at this [web page](#):

1. Create the intermediate image, as shown in Example 3-6.

Example 3-6 Start the dummy Git container

```
admin@sc74cn11:~/redbooks$ docker run --name dummy-git -it --entrypoint /bin/bash
-d ubuntu
c0423231dc39987cd3e2fb8b37216b303d52d56f763ffe2481dfb1a13637f248
```

2. After the image is up and running, log on to the image, as shown in Example 3-7.

Example 3-7 Log on to the dummy Git container

```
admin@sc74cn11:~/redbooks$ docker exec -it dummy-git bash
root@c0423231dc39:/#
```

3. Install the net-tools package, which includes important tools for controlling the network subsystem of the Linux kernel, including arp, hostname, ifconfig, netstat, rarp, and route, as shown in Example 3-8.

Example 3-8 Install the required tools in the dummy Git container

```
root@c0423231dc39:/# apt update
Get:1 http://ports.ubuntu.com/ubuntu-ports bionic InRelease [242 kB]
Get:2 http://ports.ubuntu.com/ubuntu-ports bionic-updates InRelease [88.7 kB]
```

<output omitted intentionally>

```
Get:17 http://ports.ubuntu.com/ubuntu-ports bionic-security/restricted s390x
Packages [581 B]
Get:18 http://ports.ubuntu.com/ubuntu-ports bionic-security/multiverse s390x
Packages [1849 B]
Fetched 15.7 MB in 5s (3485 kB/s)
Reading package lists... Done
Building dependency tree
Reading state information... Done
4 packages can be upgraded. Run 'apt list --upgradable' to see them.
```

```
root@c0423231dc39:/# apt install -y git net-tools vim wget
Reading package lists... Done
Building dependency tree
Reading state information... Done
The following additional packages will be installed:
  ca-certificates file git-man krb5-locales less libasn1-8-heimdal libbsd0
  libcurl3-gnutls libedit2 liberror-perl libexpat1 libgdbm-compat4 libgdbm5
```

```
libgpm2 libgssapi-krb5-2 libgssapi3-heimdal libhcrypto4-heimdal  
libheimbase1-heimdal libheimntlm0-heimdal libhx509-5-heimdal libk5crypto3  
libkeyutils1
```

<output omitted intentionally>

```
Setting up xauth (1:1.0.10-1) ...  
Setting up libldap-2.4-2:s390x (2.4.45+dfsg-1ubuntu1.5) ...  
Setting up libcurl3-gnutls:s390x (7.58.0-2ubuntu3.8) ...  
Setting up git (1:2.17.1-1ubuntu0.7) ...  
Processing triggers for libc-bin (2.27-3ubuntu1) ...  
Processing triggers for ca-certificates (20190110~18.04.1) ...  
Updating certificates in /etc/ssl/certs...  
0 added, 0 removed; done.  
Running hooks in /etc/ca-certificates/update.d...  
done.
```

-
4. After the prerequisite tools are installed, continue to download (clone) the Git repository, as shown in Example 3-9.

Example 3-9 Clone the git repository

```
root@c0423231dc39:/# git clone https://github.com/ot4i/ace-docker.git  
Cloning into 'ace-docker'...  
remote: Enumerating objects: 2340, done.  
remote: Total 2340 (delta 0), reused 0 (delta 0), pack-reused 2340  
Receiving objects: 100% (2340/2340), 3.22 MiB | 19.86 MiB/s, done.  
Resolving deltas: 100% (897/897), done.
```

5. After the download completes successfully, change into the newly created directory to:

```
cd ace-docker
```

Example 3-10 lists all the files that should be downloaded from the Git repository.

Example 3-10 Verify the downloaded git files

```
root@c0423231dc39:/ace-docker# ls -l  
total 208  
-rw-r--r-- 1 root root 1881 Jun 14 11:17 CHANGELOG.md  
-rw-r--r-- 1 root root 83670 Jun 14 11:17 LICENSE  
-rw-r--r-- 1 root root 22365 Jun 14 11:17 README.md  
-rw-r--r-- 1 root root 533 Jun 14 11:17 ace_compileBars.sh  
-rw-r--r-- 1 root root 1439 Jun 14 11:17 ace_config_agent.sh  
-rw-r--r-- 1 root root 563 Jun 14 11:17 ace_configBars.sh  
-rw-r--r-- 1 root root 704 Jun 14 11:17 ace_config_extensions.sh  
-rw-r--r-- 1 root root 2392 Jun 14 11:17 ace_config_keystore.sh  
-rw-r--r-- 1 root root 1504 Jun 14 11:17 ace_config_logging.sh  
-rw-r--r-- 1 root root 733 Jun 14 11:17 ace_config_odbcini.sh  
-rw-r--r-- 1 root root 1020 Jun 14 11:17 ace_config_policy.sh  
-rw-r--r-- 1 root root 676 Jun 14 11:17 ace_config_serverconf.sh  
-rwxr-xr-x 1 root root 1837 Jun 14 11:17 ace_config_setdbparms.sh  
-rw-r--r-- 1 root root 724 Jun 14 11:17 ace_config_ssl.sh  
-rw-r--r-- 1 root root 1409 Jun 14 11:17 ace_config_truststore.sh  
-rw-r--r-- 1 root root 3912 Jun 14 11:17 ace_config_webusers.sh  
-rw-r--r-- 1 root root 1030 Jun 14 11:17 ace_discover_port_overrides.sh  
-rw-r--r-- 1 root root 378 Jun 14 11:17 ace_env.sh  
-rwxr-xr-x 1 root root 740 Jun 14 11:17 ace_integration_server.sh
```

```

-rw-r--r-- 1 root root 1454 Jun 14 11:17 ace_license_check.sh
-rw-r--r-- 1 root root 386 Jun 14 11:17 ace_mqsicommand.sh
-rw-r--r-- 1 root root 11691 Jun 14 11:17 app_connect_light_256x256.png
-rw-r--r-- 1 root root 4383 Jun 14 11:17 appconnect_enterprise_logo.svg
drwxr-xr-x 1 root root 36 Jun 14 11:17 asoc
-rwxr-xr-x 1 root root 1638 Jun 14 11:17 build-rhel.sh
drwxr-xr-x 1 root root 72 Jun 14 11:17 cmd
drwxr-xr-x 1 root root 42 Jun 14 11:17 deps
drwxr-xr-x 1 root root 152 Jun 14 11:17 experimental
drwxr-xr-x 1 root root 56 Jun 14 11:17 internal
drwxr-xr-x 1 root root 44 Jun 14 11:17 licenses
drwxr-xr-x 1 root root 186 Jun 14 11:17 sample
drwxr-xr-x 1 root root 188 Jun 14 11:17 ubi
drwxr-xr-x 1 root root 56 Jun 14 11:17 vendor

```

The downloaded files are used for input to build the ACE Docker image. The files must be copied into the underlying zCX instance by running Docker commands. To do this, leave the intermediate dummy-git container by running the **exit** command, as shown in Example 3-11.

Example 3-11 Exit from container

```

root@7901655109e3:/# exit
exit

```

6. Run the following command from the zCX image to copy the files into the local zCX image:
`docker cp dummy-git:/ace-docker`

3.2.3 Building the Dockerfile for the ACE image

Complete the following steps to build the ACE image:

1. Copy the ACE binary file (11.0.0-ACE-LINUXZ64-FP0010.tar.gz) by running the command that is shown in Example 3-12 to the ace-docker/deps folder.

Example 3-12 Copy ACE binary file

```

admin@sc74cn11:~/redbooks$ cp 11.0.0-ACE-LINUXZ64-FP0010.tar.gz
ace-docker/deps/

admin@sc74cn11:~/redbooks$ ls -la ace-docker/deps/
total 706616
drwxr-xr-x 1 admin admin 108 Sep 10 18:53 .
drwxr-xr-x 1 admin admin 1170 Sep 10 18:01 ..
-rw-r--r-- 1 admin admin 112 Sep 10 18:01 .gitignore
-rw-r----- 1 admin admin 723570314 Sep 10 18:53
11.0.0-ACE-LINUXZ64-FP0010.tar.gz
drwxr-xr-x 1 admin admin 26 Sep 10 18:01 OpenTracing
admin@sc74cn11:~/redbooks$

```

2. Download other binaries from the Docker hub.

Important Note: To download files directly from the Docker hub, you must have a Docker ID. An ID can be created at [this website](#).

If the download is attempted without a valid logon, an error is shown, as shown in Example 3-13.

Example 3-13 Error when attempting a build directly from Docker without a valid logon ID.

```
admin@sc74cn11:~/redbooks$ docker build -t aceonly-img -f  
ace-docker/ubi/Dockerfile.aceonly .  
Sending build context to Docker daemon 32.34MB  
Step 1/43 : FROM golang:1.10.3 as builder  
Get https://registry-1.docker.io/v2/library/golang/manifests/1.10.3: unauthorized:  
incorrect username or password
```

To build the image without the authentication error, log on before the Docker build command is run, as shown in Example 3-14.

Example 3-14 Successful download from docker

```
admin@sc74cn11:~/redbooks$ docker login  
Login with your Docker ID to push and pull images from Docker Hub. If you don't  
have a Docker ID, head over to https://hub.docker.com to create one.  
Username: <your_user_name>  
Password:  
WARNING! Your password will be stored unencrypted in  
/home/admin/.docker/config.json.  
Configure a credential helper to remove this warning. See  
https://docs.docker.com/engine/reference/commandline/login/#credentials-store  
  
Login Succeeded
```

3. After logging in to Docker, the Dockerfile that was downloaded from the Git repository can be adjusted for zCX.

Edit the file by using the vi editor, as shown in Example 3-15.

Example 3-15 Adjust the downloaded file, dockerfile.aceonly

```
admin@sc74cn11:~/redbooks$ vi ace-docker/ubi/Dockerfile.aceonly
```

If you do not want to specify the name of the ACE installation binaries on every build, (the file that is shown in Example 3-2, “List downloaded file” on page 110), you can modify the lines that are shown in **bold** in Example 3-16 and change them to the new lines that are shown in non-bold.

Example 3-16 Specify the name of the ACE installation binaries.

```
ARG ACE_INSTALL=ace-11.0.0.2.tar.gz  
ARG ACE_INSTALL=11.0.0-ACE-LINUXZ64-FP0010.tar.gz
```

If you want to add utilities to the Docker container, you can modify the lines that are shown in **bold** in Example 3-17 and change them to the new lines that are shown in non-bold.

Example 3-17 Adding utilities to the Docker container

```
RUN microdnf update && microdnf install findutils util-linux unzip python3 &&  
microdnf clean all  
RUN microdnf update && microdnf install findutils tar git wget vim openssl  
util-linux unzip python3 && microdnf clean all
```

4. After editing the file, close the edit session by running the **:wq** command.

3.2.4 Building the ACE Docker image

After the Dockerfile is edited and saved successfully, the Docker image is downloaded from IBM Support and can be built, as shown in Example 3-18.

Example 3-18 Build the ACE Docker image

```
admin@sc74cn11:~/redbooks$ cd /home/admin/redbooks/ace-docker/  
admin@sc74cn11:~/redbooks/ace-docker$  
  
admin@sc74cn11:~/redbooks/ace-docker$ docker build -t aceonly-img -f  
ubi/Dockerfile.aceonly . Use this only if you followed Example 2-16, otherwise use  
the command shown in Example 3-19  
Sending build context to Docker daemon 32.34MB  
Step 1/43 : FROM golang:1.10.3 as builder  
----> 1311442b6183  
Step 2/43 : WORKDIR /go/src/github.com/ot4i/ace-docker/  
----> Using cache  
----> 24565e9ee58d  
  
<output omitted intentionally>  
  
Step 43/43 : ENTRYPOINT ["runaceserver"]  
----> Running in 461a644cf95a  
Removing intermediate container 461a644cf95a  
----> bd26c4fa8b62  
Successfully built bd26c4fa8b62  
Successfully tagged aceonly-img:latest
```

If you did not specify the ACE installation binaries file that is shown in Example 3-19, you must add an argument to the build command to specify them.

Example 3-19 Replacement build command

```
admin@sc74cn11:~/redbooks/ace-docker$ docker build --build-arg  
ACE_INSTALL=11.0.0-ACE-LINUXZ64-FP0010.tar.gz -t aceonly-img -f  
ubi/Dockerfile.aceonly
```

After building the image, the ACE server can be started by running the following command:

```
docker run -d --name aceserver-app-1 -p 7600:7600 -p 7800:7800 -p 7843:7843 -v  
ace-only-data:/home/aceuser/ace-server --env LICENSE=accept --env  
ACE_SERVER_NAME=ACESERVER1 aceonly-img
```

You see the following output by running the previous command, as shown in Example 3-20.

Example 3-20 Output of docker run

```
admin@sc74cn11:~/redbooks/ace-docker$ docker run -d --name aceserver-app-1 -p
7600:7600 -p 7800:7800 -p 7843:7843 -v ace-only-data:/home/aceuser/ace-server
--env LICENSE=accept --env ACE_SERVER_NAME=ACESERVER1 aceonly-img
ff2ddfe21ce94cf11890c391dd18ce647ace4a8c6131353e6441030ea81c01e8
```

Verify the status of the server by running the command that is shown in Example 3-21. You know that the server is up and running as indicated by the server that it is listening on port 7600.

Example 3-21 Status the ACE docker container

```
admin@sc74cn03:~/redbooks/ace-docker$ docker logs aceserver-app-1
2020-09-10T19:25:06.447Z Image created: 2020-09-10T18:31:20+00:00
2020-09-10T19:25:06.447Z Image revision: Not specified
2020-09-10T19:25:06.447Z Image source: Not specified
2020-09-10T19:25:06.594Z ACE version: 11009
2020-09-10T19:25:06.594Z ACE level: S000-L200527.16701
2020-09-10T19:25:06.594Z ACE build type: Production, 64 bit, s390x_linux_2
2020-09-10T19:25:06.594Z Checking for valid working directory
2020-09-10T19:25:06.594Z Checking if work dir is already initialized
2020-09-10T19:25:06.594Z Checking for contents in the work dir
2020-09-10T19:25:06.594Z Work dir initialization complete
2020-09-10T19:25:06.594Z Performing initial configuration of integration server
2020-09-10T19:25:06.595Z No content server url available
2020-09-10T19:25:06.595Z Initial configuration of integration server complete
2020-09-10T19:25:06.595Z Discovering override ports
2020-09-10T19:25:06.600Z Successfully discovered override ports
2020-09-10T19:25:06.600Z Starting integration server
2020-09-10T19:25:06.600Z No default application name supplied. Using the
integration server name instead.
2020-09-10T19:25:06.600Z Waiting for integration server to be ready
2020-09-10T19:25:06.604Z Integration server not ready yet
.....2020-09-10 19:25:06.746787: .2020-09-10 19:25:06.746905: Integration server
'ACESERVER1' starting initialization; version '11.0.0.9' (64-bit)
.....2020-09-10 19:25:09.164314: IBM App
Connect Enterprise administration security is inactive.
2020-09-10 19:25:09.172017: The HTTP Listener has started listening on port '7600'
for 'RestAdmin http' connections.
2020-09-10 19:25:09.185685: Integration server has finished initialization.
2020-09-10T19:25:11.611Z Integration server is ready
2020-09-10T19:25:11.612Z Metrics are disabled
```

The ACE Web UI is accessible by way of `http://<your_zcx_instance_name>:7600`.

Figure 3-7 shows the web user interface for the ACE Server.

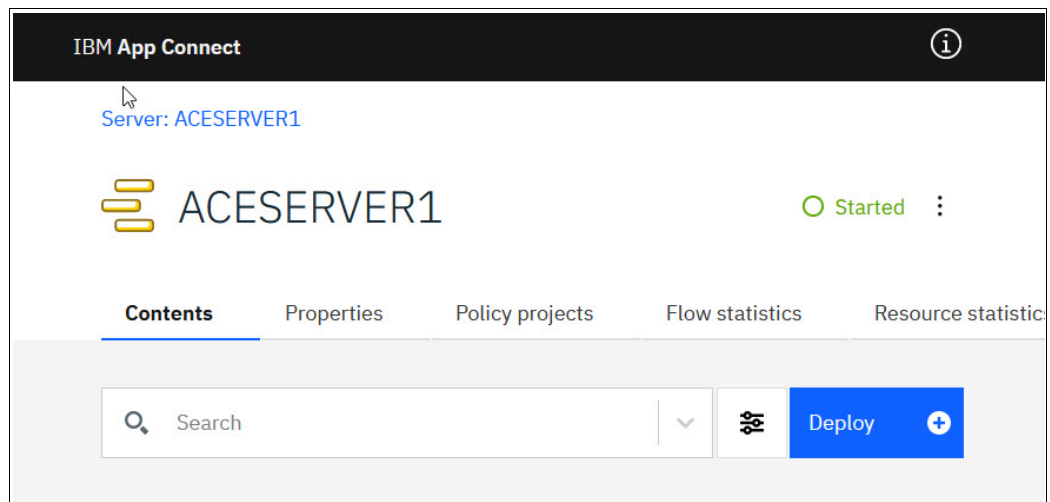


Figure 3-7 UI from the running ACE instance

To stop the running container, run the **docker stop aceserver-app-1** command.

The container shuts down cleanly, which stops the integration server.

3.3 Configuration details

After installing ACE on zCX, our configuration appears as shown in Figure 3-8. In our configuration, our CICS region happens to be in another LPAR, but your CICS region can be located anywhere, including in the same LPAR as your zCX.

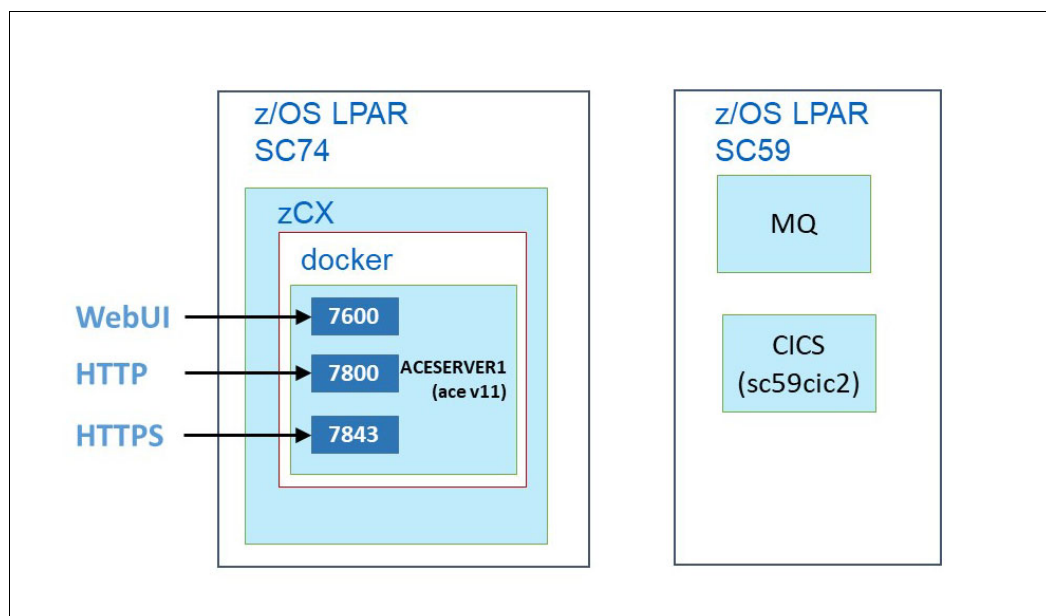


Figure 3-8 zCX configuration

Our configuration includes the following details:

- ▶ Integration server name: ACESERVER1
- ▶ Listener port for administration (including Web UI): 7600
- ▶ HTTP listener port: 7800
- ▶ HTTPS listener port: 7843

We use this integration server for our use case scenario. We deploy an application that receives a REST API call and starts a CICS program to query a customer and send the response back to the REST client.

3.4 Deploying an application to ACE to integrate with CICS

The intent of the use case in this section is to show the deployment of an application to the ACE integration server in zCX. The application is a message flow that accesses a CICS application on the z/OS system. The CICS application is a simple one that returns a fictional customer address that is based on a customer number 1 - 10 as input.

The use case is shown in Figure 3-9.

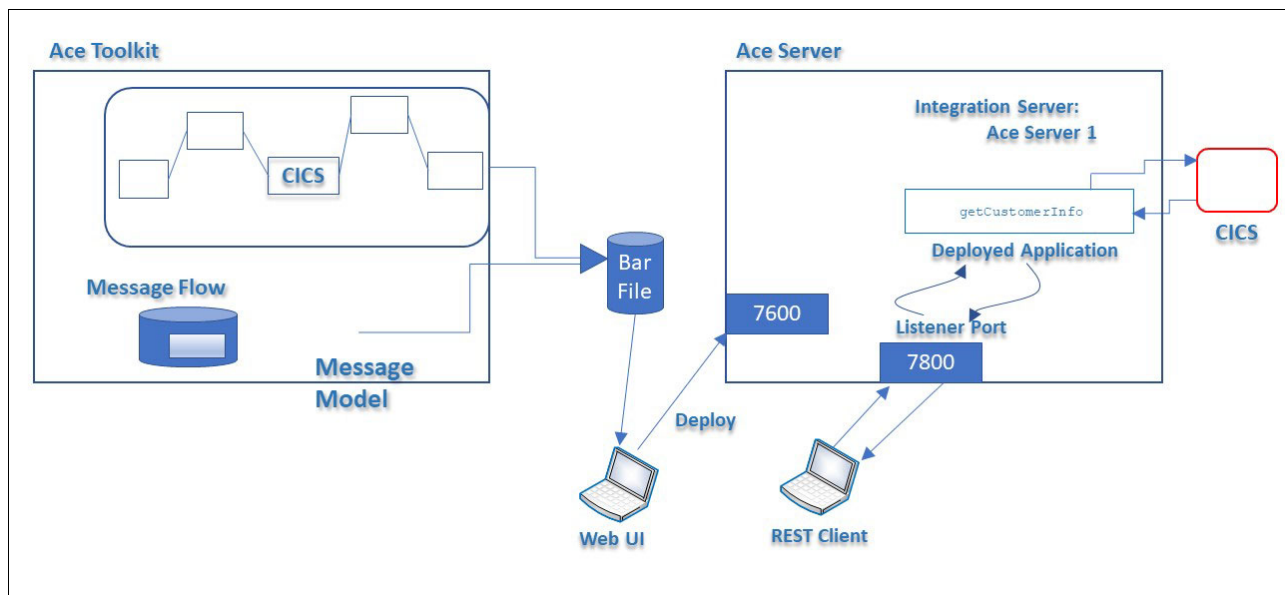


Figure 3-9 ACE use case

This use case includes the following key points:

- ▶ Message flow:
 - Built in the ACE toolkit
 - Accesses a CICS COBOL program CUSTINQ
 - Exposed as a REST API in port 7800 for external clients.
- ▶ CICS program expects data in COPYBOOK format
- ▶ Incoming REST input data is in JSON format
- ▶ Output to the REST client is in JSON format

Message flow

The message flow is built in the ACE toolkit. We do not describe how to build the message flow in this book. The message flow looks similar to the example that is shown in Figure 3-10.

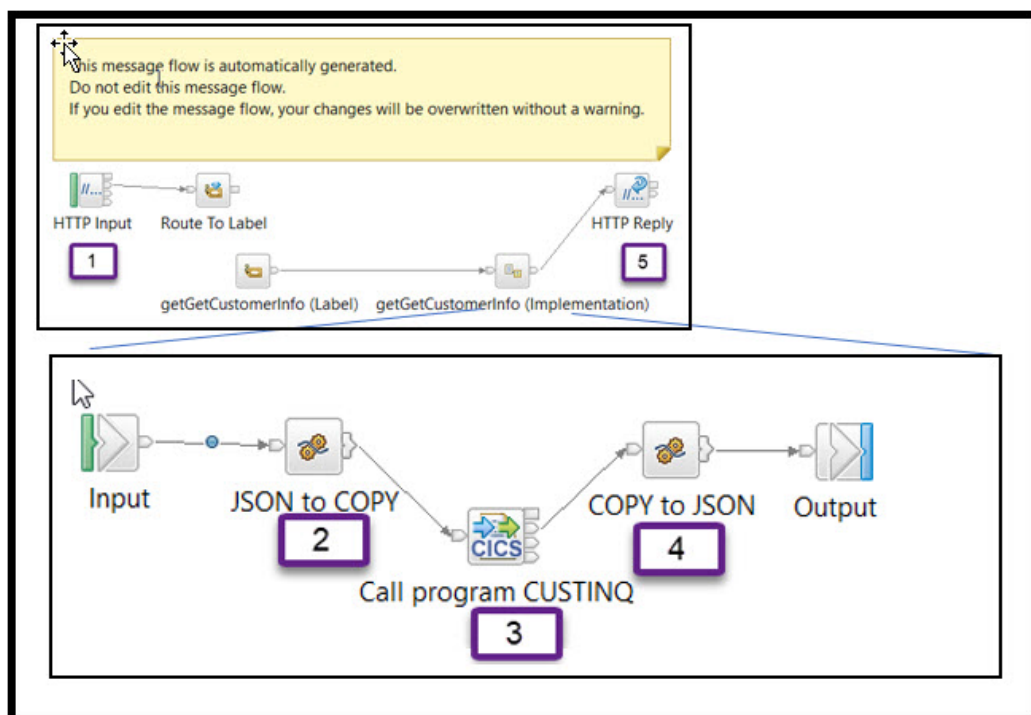


Figure 3-10 Message flow for use case

The message flow conducts the following process:

1. Receives the REST call.

It identifies the flow as a REST API with a query parameter that accepts the customer number. The input data to the flow from the REST client is in JSON format.

2. Converts JSON input data to COPYBOOK format.

In this step, the JSON format is converted to a copybook format and the input customer number is passed to the CustNo field of the record. The copybook is what the CICS program expects in the COMMAREA. The copybook format is shown in Example 3-22.

Example 3-22 Copybook for CICS program

```

01 DFHCOMMAREA.
   02 CustNo      PIC S9(9) COMP-5 .
   02 LastName    PIC A(25).
   02 FirstName   PIC A(15).
   02 Address1    PIC X(20).
   02 City        PIC A(20).
   02 State       PIC A(5).
   02 Country     PIC X(15).
   02 RetCode     PIC S9.
  
```

3. Starts the CICS program.

The message flow calls the CICS program CUSTINQ from the CICS node with the parameters that are shown in Figure 3-11. Port 3001 is the IPIC listener port in CICS.

CICS Request Node Properties - Call program CUSTINQ	
Description	
Basic	CICS server* <input type="text" value="tcp://129.40.23.55:3001"/>
Transactionality	<i>e.g. tcp://cicsserver.com:11111 or CICSConnection</i>
Request	Program name* <input type="text" value="CUSTINQ"/>
Result	Data structure <input checked="" type="radio"/> Commarea <input type="radio"/> Channel
Response Message Parsing	Commarea length* <input type="text" value="105"/>
Parser Options	
Validation	Security identity <input type="text" value=" <The name of an identity defined with mqsisetd"/>
Monitoring	Request timeout (secs) <input type="text" value="120"/>
	Mirror transaction ID <input type="text"/>
	Set EIBTRNID only <input type="checkbox"/>

Figure 3-11 CICS node properties

4. Converts from COPYBOOK to JSON format.

In this step, the response from CICS is converted from COPYBOOK format to JSON format.

5. Replies with response back to the REST client.

The JSON response is sent back to the REST client that made the request.

3.4.1 Deploying to ACE run time in zCX

To deploy a solution to an environment, you can package the resources into a Broker Archive (BAR) file. You can add applications and libraries to the BAR file. The BAR file is the deployment artifact to the ACE integration server.

Complete the following steps:

1. Create the BAR file.

Use the ACE Toolkit to build the BAR file. The BAR file contains the REST API application that we built and the message model of the COPYBOOK structure, as shown in Figure 3-12.

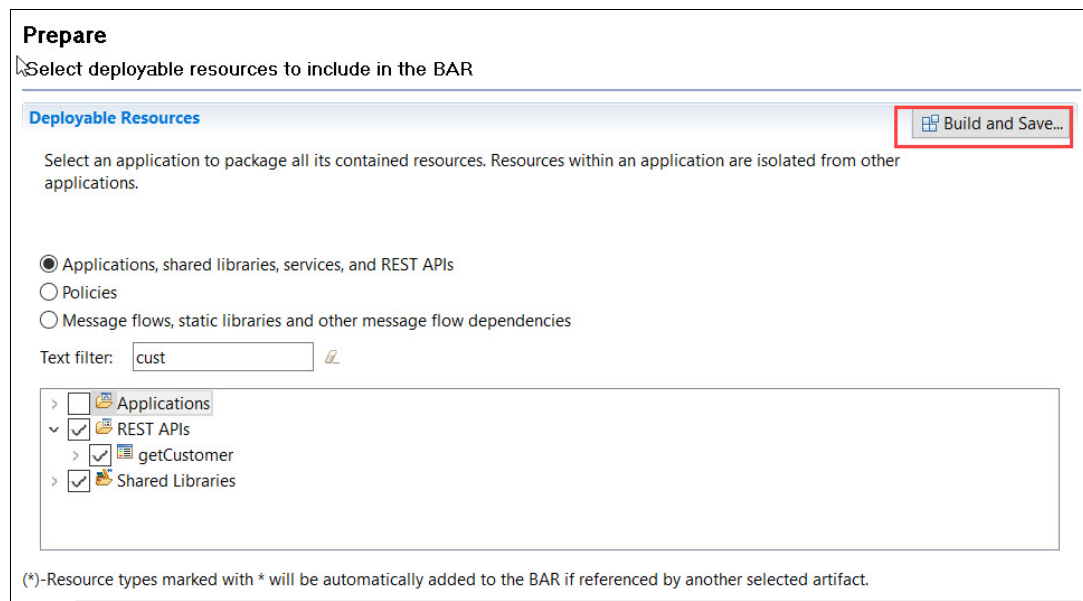


Figure 3-12 Build and save BAR file

The generated BAR file is saved as `getCustomerApp.bar` on the desktop, as shown in Figure 3-13.

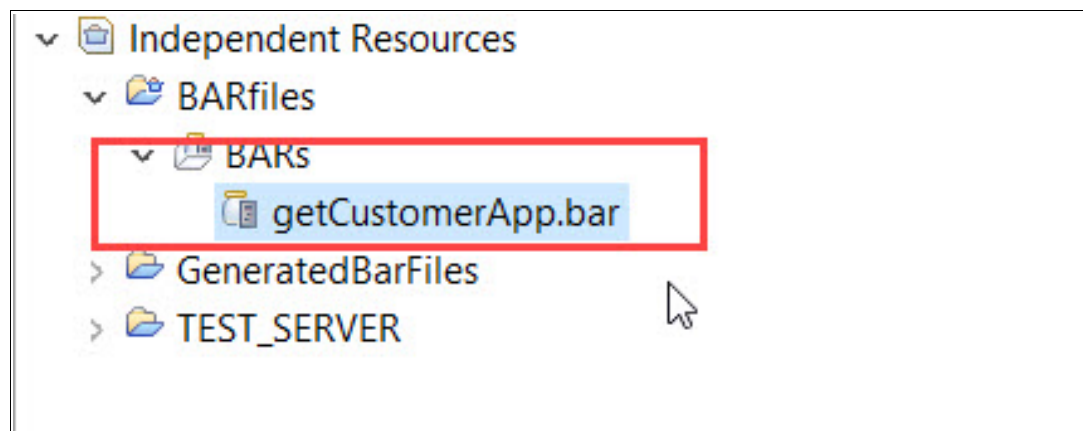


Figure 3-13 Generated BAR file

2. Open the Web UI in your web browser and use listener port 7600:
`http://<your zcx instance>:7600`
The Web UI starts with no deployments, as shown in Figure 3-14.

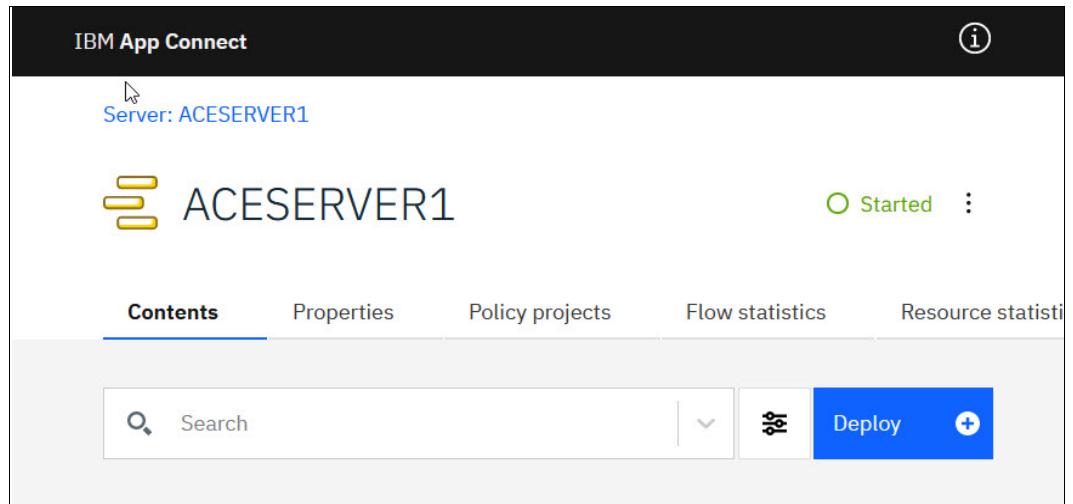


Figure 3-14 WEb UI of ACE Server

3. Deploy the BAR file that was created to ACESERVER1 by clicking **Deploy**.
4. Add the BAR file, `getCustomerApp.bar`, that was created, as shown in Figure 3-15.

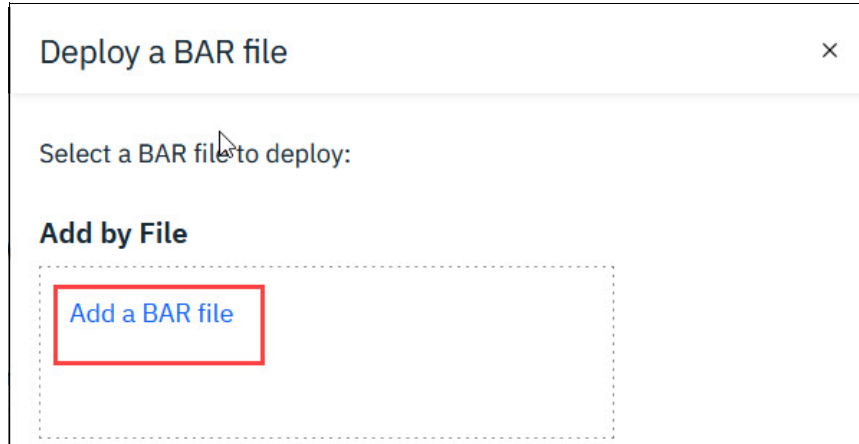


Figure 3-15 Add BAR file

5. Deploy the BAR file by clicking **Deploy**, as shown in Figure 3-16.

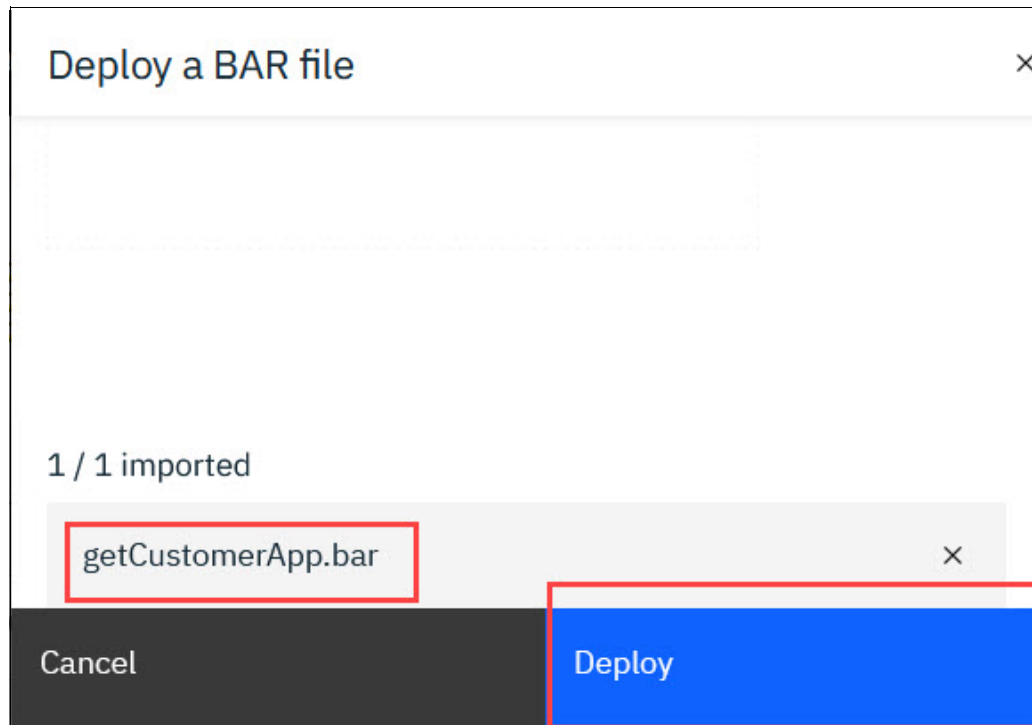


Figure 3-16 Deploy BAR file

After a successful deployment, the API and the message model are visible on the Web UI, as shown in Figure 3-17.

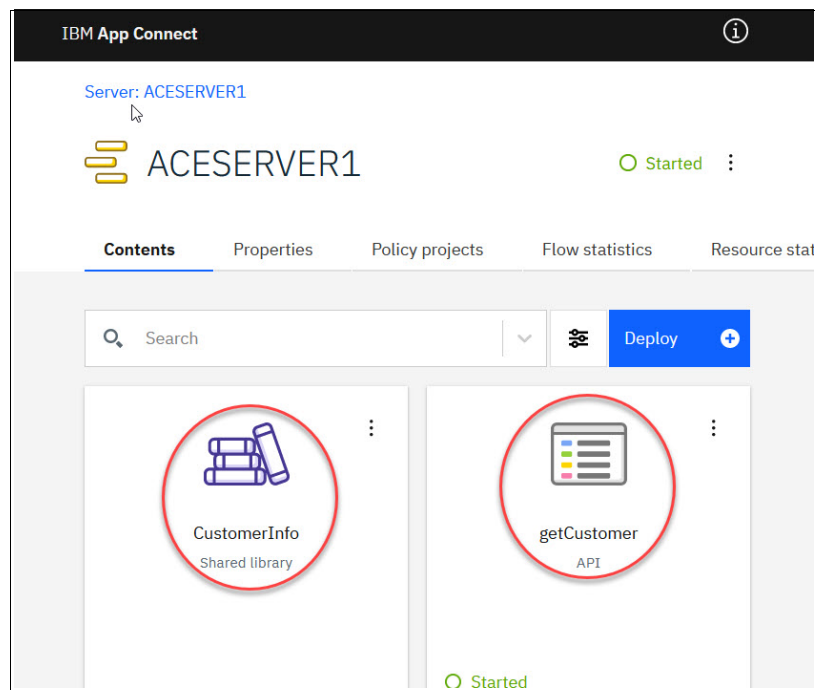


Figure 3-17 Deployed artifacts

3.4.2 Using the Web UI to test deployed REST APIs

The REST API is made available by way of port 7800. Starting from ACEv11.0.0.9, ACE is providing a new dedicated component for helping to test your deployed REST APIs.

Complete the following steps to perform the test:

1. On the Web UI, click the **getCustomer** API.
2. We see that the REST API was deployed to an integration server. Click the available operation **GET /getCustomerInfo** and switch to the Try it tab. Enter 9 in the **custno*** field and click **Send**, as shown in Figure 3-18.

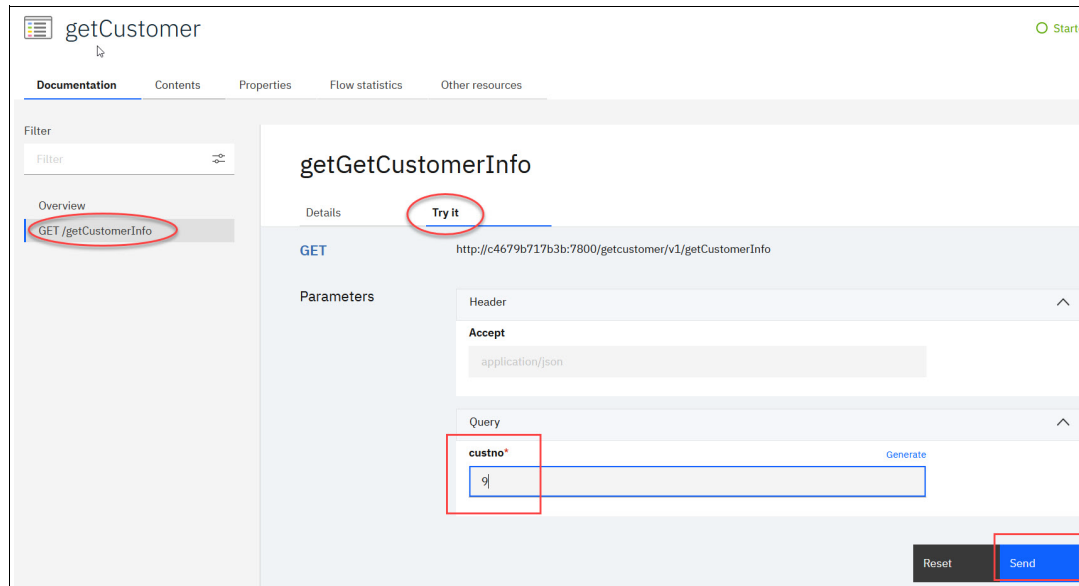


Figure 3-18 Try the REST API

For this example, an integration server with default settings was used. When we attempted to start the REST API, we received the error, as shown in Figure 3-19. This error was most likely caused by a need to enable the integration server to permit cross-origin requests from a web browser by enabling Cross-Origin Resource Sharing (CORS). Many control options are available, depending on how open you want our CORS settings to be set up.

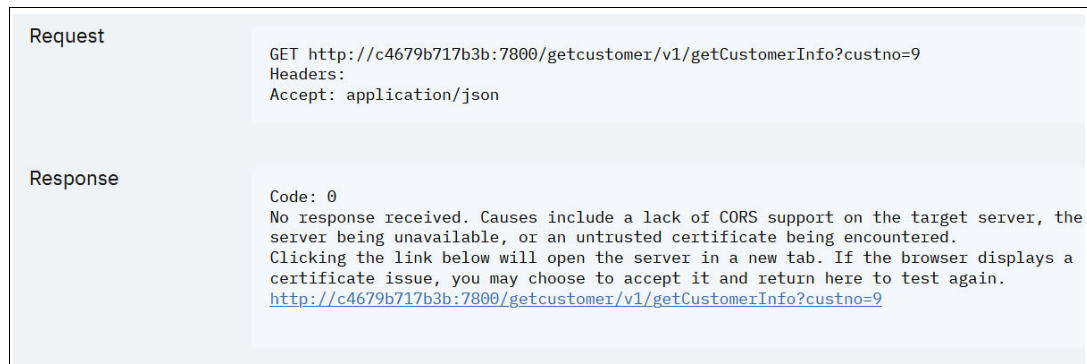


Figure 3-19 No Response from REST API

Set one of the open set of configuration options in the integration server's `server.conf.yaml` file, `CORSEnabled`, to true and restart the server.

3. Run an interactive bash shell on the ACE container by running the command that is shown in Example 3-23.

Example 3-23 Running interactive bash

```
admin@sc74cn09:~$ docker exec -it aceserver-app-1 bash
```

4. Dynamically configure the ACE integration server. Create a directory that is named `serverconf` and make a copy of the `server.conf.yaml` file, as shown in Example 3-24.

Example 3-24 Creating a directory and making a back up copy of the configuration file

```
[aceuser@c4679b717b3b ~]$ cd initial-config/
[aceuser@c4679b717b3b initial-config]$ mkdir serverconf
[aceuser@c4679b717b3b initial-config]$ cd serverconf
[aceuser@c4679b717b3b serverconf]$ cp /home/aceuser/ace-server/server.conf.yaml
server.conf.yaml
```

5. Edit the `server.conf.yaml` file by using the `vi` editor and uncomment and set `CORSEnabled` to `true` in the `HTTPConnector` section, as shown in Example 3-25.

Example 3-25 Setting CORSEnabled

```
[aceuser@c4679b717b3b serverconf]$ vi server.conf.yaml
```

```
HTTPConnector:
  #ListenerPort: 0                # Set non-zero to set a specific port,
  defaults to 7800
  #ListenerAddress: '0.0.0.0'    # Set the IP address for the listener to
  listen on. Default is
  #AutoRespondToHTTPHEADRequests: false # Automatically respond to HTTP HEAD
  #ServerName: ''                # Set the value to be returned in the
  'Server' HTTP header.
  CORSEnabled: true             # Set the value to true to make the listener
  respond to valid
  #CORSAllowOrigins: '*'
  #CORSAllowCredentials: false
  #CORSExposeHeaders: 'Content-Type'
  #CORSMaxAge: -1
  #CORSAllowMethods: 'GET,HEAD,POST,PUT,PATCH,DELETE,OPTIONS'
  #CORSAllowHeaders: 'Accept,Accept-Language,Content-Language,Content-Type'
```

6. Exit from the bash shell by running the `exit` command, as shown in Example 3-26.

Example 3-26 Exit from container bash shell

```
[aceuser@c4679b717b3b ~]$ exit
```

7. Stop and start the container to make the configuration take effect, as shown in Example 3-27.

Example 3-27 Stop and start container

```
admin@sc74cn09:~$ docker stop aceserver-app-1
aceserver-app-1
admin@sc74cn09:~$ docker start aceserver-app-1
aceserver-app-1
```

8. Open the browser and point to the following URL:

`http://<your zcx>:7800/getcustomer/v1/getCustomerInfo?custno=9`

The response from CICS provides information about customer number 9, as shown in Figure 3-20.



Figure 3-20 Response from CICS



IBM Aspera fasp.io Gateway

The IBM Aspera® fasp.io Gateway is a lightweight TCP/IP tunnel for high-speed bidirectional data transport. It uses the IBM Fast and Secure Protocol (IBM FASP®), which is efficient when transferring data over high latency networks, or those that tend to lose packets.

From IBM MQ Version 9.2.0 (Long-Term Release), customers with entitlement to IBM MQ Advanced for z/OS VUE are provided with free entitlement and access to the gateway for transferring IBM MQ messages between IBM MQ Advanced for z/OS VUE queue managers. Alternatively, customers can purchase a separate entitlement to the gateway.

This chapter describes about how to deploy the IBM Aspera fasp.io Gateway with IBM MQ on a zCX container and includes the following topics:

- ▶ 4.1, “Introduction to Aspera FASP.io Gateway” on page 130
- ▶ 4.2, “Aspera configuration details” on page 131
- ▶ 4.3, “Integration with IBM MQ Advanced for z/OS, VUE” on page 141

4.1 Introduction to Aspera FASP.io Gateway

In this age of digital economy era, fast services and connectivity become an important factor to cloud and native applications today.

The IBM Aspera fasp.io Gateway is a software solution that provides significant improvements in performance and service quality when transferring data between highly remote or dispersed locations in unfavorable network conditions, such as high latency and packet loss. Fast and Secure Protocol (FASP) provides significant improvements in performance and service quality. IBM MQ queue managers can take advantage of it and provide better service.

This technology was designed to deliver 100% bandwidth efficient transport of bulk data over any IP network. The IBM Aspera fasp.io Gateway uses an efficient algorithm for fast data transfer to retransmit only needed data, obtaining good performance independent of network delay or packet loss. It achieves speeds up to hundreds of times faster than FTP and HTTP that enables maximum speed without network saturation.

To ensure exceptional security for business-critical digital assets, Aspera uses open standards cryptography for user authentication, data encryption, and data-integrity verification.

The [IBM Aspera File Transfer Calculator](#) can be used to estimate the gains of using Aspera versus TCP.

The IBM Aspera fasp.io Gateway delivers the following key features and capabilities:

- ▶ Extraordinary bandwidth control
The intelligent adaptive transmission-rate control mechanism in the FASP protocol enables fast, automatic discovery of available bandwidth to fully leverage capacity while being fair to other traffic.
- ▶ Flexible and open architecture
Aspera supports interoperable file and directory transfers between all major operating systems and cloud platforms. It also provides a complete, modern software API.
- ▶ Software-only design
Aspera is a software-based technology that can run on commodity hardware and over standard, unmodified IP networks, which makes for easier, more scalable deployments and lower total cost of ownership.

The Aspera Gateway component can be integrated quickly and easily with existing applications that use a TCP connection for data flow. By achieving per-process aggregate bandwidths as high as 2.5 GBps (regardless of distance and network conditions), Aspera fasp.io Gateway outperforms TCP-based data flows over wide area networks that exhibit high round-trip times and high packet loss.

The Gateway uses standard UDP in the transport layer and achieves decoupled congestion and reliability control in the application layer through a theoretically optimal approach that retransmits precisely the real packet loss on the channel

Figure 4-1 shows an example in which a messaging service, such as IBM MQ or Event Streams, can be used with fasp.io.

Note: The connection between the fasp.io gateways containers that use the FASP protocol is User Datagram Protocol (UDP), not Transmission Control Protocol (TCP).

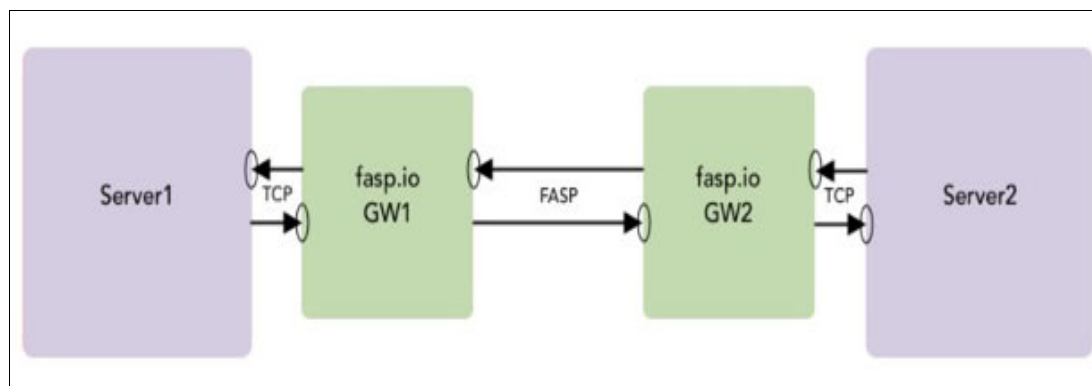


Figure 4-1 IBM MQ or Event Streams using fasp.io

4.2 Aspera configuration details

In our environment, we defined IBM MQ sender-receiver channels between two queue managers, MQZ1 and MQZ2, each running on separate z/OS LPARs: wtsc74 and wtsc75.

We used the information that is listed in Table 4-1 in our configuration.

Table 4-1 Lab configuration

LPAR	wtsc74	wtsc75
Queue Manager	MQZ1	MQZ2
FQDN / IP address	wtsc74.pbm.ihost.com 129.40.23.1	wtsc75.pbm.ihost.com 129.40.23.3
Queue Manager Listener Port	1414	2414
zCX Name	sc74cn04.pbm.ihost.com	sc74cn09.pbm.ihost.com
zCX IP address	129.40.23.71	129.40.23.76
Fasp.IO container name	faspio-app-sc74cn04	faspio-app-sc74cn09
Fasp.IO port (UDP)	2414	1414

As noted in Table 4-1, we preferred to make available different TCP/UDP ports on each zCX container. You might want to do the same or define your own ports.

Note: In our instructions, we appended the zCX name to the container name, for example:

- For the sc74cn04 zCX server, the fasp.io container name is faspio-app-sc74cn04
- For the sc74cn09 zCX server, the fasp.io container name is faspio-app-sc74cn09

We installed fasp.io in two parts:

- ▶ “Configuration Part 1: fasp.io on the first zCX (sc74cn09) instance”
- ▶ “Configuration Part 2: fasp.io on the second zCX (sc74cn04) instance” on page 137

4.2.1 Configuration Part 1: fasp.io on the first zCX (sc74cn09) instance

In this configuration, we obtain the Aspera Gateway from an IBM MQ installation. Use of the Aspera gateway that is provided with IBM MQ is limited to IBM MQ messages, unless the gateway is separately entitled. To use the IBM MQ-provided Aspera gateway, you must have one or more of the following entitlements:

- ▶ IBM MQ Advanced for Multiplatforms
- ▶ IBM MQ Appliance
- ▶ IBM MQ Advanced for z/OS VUE

For IBM MQ Advanced for z/OS VUE entitlement, you can get the Aspera gateway from the Connector Pack component that is part of the SMP/E installation. The Connector Pack component is identified by the following information:

- ▶ FMID: HAV9110
- ▶ COMPID: 5655AV100
- ▶ Component Name: IBM MQ Connector Pack for z/OS

Note: For more information, see the IBM MQ Advanced for z/OS VUE program directory PDF files that are available at [IBM Knowledge Center](#).

Complete the following steps to configure our first zCX instance. After the SMP/E installation was complete, which includes the Aspera gateway from the Connector Pack component, the CSQ8FSP1.tar.Z file is available in the following UNIX System Services directory:

/usr/lpp/mqm/V9R1MX:

1. Download the CSQ8FSP1.tar.Z as a file named CSQ8FSP1.tar to a directory of your choice.
2. On the sc74cn09 zCX server, create the /home/admin/faspio directory:

```
mkdir -p /home/admin/faspio
cd /home/admin/faspio
```

3. Upload the file CSQ8FSP1.tar into the /home/admin/faspio directory.
4. To extract the required installation files, complete the following steps:

- a. Create a dummy container to decompress the files by running the following command:

```
docker run --name dummy -it --entrypoint /bin/bash -d fedora
```

- b. Copy the CSQ8FSP1.tar file to the dummy container:

```
docker cp CSQ8FSP1.tar dummy:.
```

- c. Run the following command in the dummy container:

```
docker exec -it dummy bash
```

- d. Run the following commands to extract the fasp.io binary and configuration files:

```
yum install -y unzip
tar xf CSQ8FSP1.tar
cd fasp
unzip ASP_FASP.IO_GW_2.5GBPSV1.0.1_ZLNX.zip
```

- e. Install ibm-fasp-io-gateway by running the following command:

```
rpm -ivh ibm-fasp.io-gateway-1.0.1-1.s390x.rpm
```

- f. Enter `exit` to leave the dummy container.
5. Run the following commands to copy the `/usr/bin/fasp.io-gateway` file and `/etc/fasp.io/` directory. They are copied to the new fasp container:

```
docker cp dummy:/usr/bin/fasp.io-gateway
docker cp dummy:/etc/fasp.io/
```
6. Create the `Dockerfile.faspio` file with the content that is shown in Example 4-1.

Example 4-1 Dockerfile.faspio

```
FROM fedora:latest

# ENV
ENV PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

# The author
LABEL maintainer="IBM Redbooks - Sample Dockerfile"

# Installing Linux Packages
RUN yum install -y gettext telnet net-tools vim procps

# Copying fasp.io files
RUN mkdir /etc/fasp.io/
COPY fasp.io-gateway /bin/
COPY fasp.io/* /etc/fasp.io/
# Entrypoint Fasp.io
ENTRYPOINT ["/bin/fasp.io-gateway"]
```

Note: We added `telnet`, `net-tools`, and `vim` packages to the container because they provide base networking utilities for Linux and allow us to query information about the services.

7. Build the new faspio container image by running the following command:

```
docker build -t faspio-img-new -f Dockerfile.faspio
```
8. After the image is built, run the following command to create a fasp.io container:

```
docker run --name faspio-app-sc74cn09 -p2414:2414 -p1414:1414/udp -d
faspio-img-new
```

Note: The FASP protocol uses UDP (not TCP). Remember to specify `udp` when making available the FASP port. If your zCX instance is behind a firewall, remember to open the firewall ports and provide the necessary information to the firewall team.

Table 4-2 lists the options we used.

Table 4-2 Docker run options

Option	Description
faspio-app-sc74cn09	Name of the fasp.io container. We appended the zCX name (sc74cn09) to make it easier to identify the container on each zCX instance.
-p2414:2414	Make available 2414 TCP port.
-p1414:1414/udp	Make available 1414 UDP port.
-d	Run container in background and print container ID.
faspio-img-new	Name of the fasp.io Docker image.

Architecturally, an Aspera gateway that is configured to use the FASP protocol sends and receives UDP traffic on the fasp port. To allow the UDP session to start, the fasp.io container on the zCX server side must make available the port UDP.

During our environment creation, we neglected to specify /udp when making available the FASP port and FASP gateway and received the error that is shown in Example 4-2.

Example 4-2 Output of Docker logs command

```
[2020-06-15 22:10:11.019] [faspio-c] [warning] 2:Have not received a packet
for 12 seconds.
[2020-06-15 22:10:23.021] [faspio-c] [warning] 2:Have not received a packet
for 24 seconds.
[2020-06-15 22:10:29.049] [gateway] [error] 2:Session error: Connection refused
[2020-06-15 22:10:29.049] [s2s] [info] 2:Stats downstream_bytes=0,
upstream_bytes=0
[2020-06-15 22:11:12.058] [faspio-c] [warning] 3:Have not received a packet
for 12 seconds.
[2020-06-15 22:11:24.059] [faspio-c] [warning] 3:Have not received a packet
for 24 seconds.
[2020-06-15 22:11:30.102] [gateway] [error] 3:Session error: Connection refused
[2020-06-15 22:11:30.102] [s2s] [info] 3:Stats downstream_bytes=0,
upstream_bytes=0
[2020-06-15 22:12:14.059] [faspio-c] [warning] 4:Have not received a packet
for 12 seconds.
[2020-06-15 22:12:26.061] [faspio-c] [warning] 4:Have not received a packet
for 24 seconds.
[2020-06-15 22:12:32.091] [gateway] [error] 4:Session error: Connection refused
[2020-06-15 22:12:32.092] [s2s] [info] 4:Stats downstream_bytes=0,
upstream_bytes=0
[2020-06-15 22:13:16.057] [faspio-c] [warning] 5:Have not received a packet
for 12 seconds.
[2020-06-15 22:13:28.057] [faspio-c] [warning] 5:Have not received a packet
for 24 seconds.
```

The communication between our zCX instances did not pass through a firewall. However, ensure that you allow TCP and UDP traffic between your zCX instances if they are behind a firewall device.

9. Stop the container:

```
docker stop faspio-app-sc74cn09
```


10. Create the `gateway.toml` file with the content that is shown in Example 4-3.

Example 4-3 gateway.toml file contents

```
[[bridge]]
  name = "MQZ1_MQZ2"
  [bridge.local]
    protocol = "tcp"
    host = "0.0.0.0"
    port = 2414
  [bridge.forward]
    protocol = "fasp"
    host = "129.40.23.71"
    port = 2414
[[bridge]]
  name = "MQZ2_MQZ1"
  [bridge.local]
    protocol = "fasp"
    host = "0.0.0.0"
    port = 1414
  [bridge.forward]
    protocol = "tcp"
    host = "wtsc74.pbm.ihost.com"
    port = 1414
```

11. Copy the `gateway.toml` file into the container by running the following command:

```
docker cp gateway.toml faspio-app-sc74cn09:/etc/fasp.io/gateway.toml
```

If you must enable `fasp.io` in debug mode for troubleshooting, create the `logging.toml` file with the content that is shown in Example 4-4.

Example 4-4 Logfile for fasp.io

```
#####
### LOGGING SETTINGS
#####

# Available Loggers:
#
# bridge: High-level logger for the bridge
# s2s: Stream-2-Stream session class logger
# fasp.io-cpp: Logger for the Asio/C++ FASP SDK
# fasp.io-c: Logger for the FASP protocol

# For more on how to configure logging, see full reference at:
https://github.com/guangie88/spdlog\_setup

# level is optional for both sinks and loggers
# level for error logging is 'err', not 'error'
# _st => single threaded, _mt => multi threaded
# syslog_sink is automatically thread-safe by default, no need for _mt suffix

# check out https://github.com/gabime/spdlog/wiki/3.-Custom-formatting
global_pattern = "%+"
```

```
# Async
[global_thread_pool]
queue_size = 8192
num_threads = 1

[[sink]]
name = "console"
type = "color_stdout_sink_mt"

[[logger]]
name = "bridge"
type = "async"
sinks = ["console"]
level = "debug"

[[logger]]
name = "s2s"
type = "async"
sinks = ["console"]
level = "debug"

[[logger]]
name = "fasp.io-cpp"
type = "async"
sinks = ["console"]
level = "debug"

[[logger]]
name = "fasp.io-c"
type = "async"
sinks = ["console"]
level = "debug"
```

12. Copy the logging.toml file into the container by running the following command:

```
docker cp logging.toml faspio-app-sc74cn09:/etc/fasp.io/logging.toml
```

Note: Because the logfile generates a high number of messages, use this file only if it is necessary.

13. Start the new container by running the following command

```
docker start faspio-app-sc74cn09
```

Complete the following steps to confirm that fasp.io is started

1. Check the container log to ensure that no issues exist by running the following command:

```
docker logs faspio-app-sc74cn09
```

The expected output is shown in Example 4-5.

Example 4-5 Container log file output

```
Loading gateway config file: /etc/fasp.io/gateway.toml
Loading logging config file: /etc/fasp.io/logging.toml
[2020-06-16 19:21:16.027] [gateway] [info] Gateway version: 1.0.1
5dfd35dde57c7ec251a2fae443f7b05f3e4922b1
```

```
[2020-06-16 19:21:16.027] [gateway] [info] Gateway config :
/etc/fasp.io/gateway.toml
[2020-06-16 19:21:16.027] [gateway] [info] Logging config :
/etc/fasp.io/logging.toml
[2020-06-16 19:21:16.027] [gateway] [info] MQZ1_MQZ2: 0.0.0.0:2414 tcp ->
fasp [129.40.23.71]:2414
[2020-06-16 19:21:16.033] [gateway] [info] MQZ2_MQZ1: 0.0.0.0:1414 fasp -> tcp
[129.40.23.1]:1414
```

2. Check for any shown container ports by running the following command:

```
docker port faspio-app-sc74cn09
```

The expected output is shown in Example 4-6.

Example 4-6 List-specific mapping for the container

```
2414/tcp -> 0.0.0.0:2414
1414/udp -> 0.0.0.0:1414
```

3. Confirm that fasp.io ports are opened by running the following command:

```
docker exec -it faspio-app-sc74cn09 bash -c "netstat -an | grep 0.0.0.0:*"

```

The expected output is shown in Example 4-7.

Example 4-7 Confirm fasp.io ports are opened

tcp	0	0 0.0.0.0:2414	0.0.0.0:*	LISTEN
udp	0	0 0.0.0.0:1414	0.0.0.0:*	

4.2.2 Configuration Part 2: fasp.io on the second zCX (sc74cn04) instance

Complete the following steps to configure the second zCX instance:

1. Repeat step 1 on page 132 - step 7 on page 133.
2. After the image is built, run the following command to create a fasp.io container:

```
docker run --name faspio-app-sc74cn04 -p1414:1414 -p2414:2414/udp -d
faspio-img-new
```

Note: The FASP protocol uses UDP (not TCP). Remember to specify udp when making available the FASP port. If your zCX instance is behind a firewall, remember to open the firewall ports and provide the necessary information to the firewall team.

Table 4-3 lists the options we used for the second zCX instance.

Table 4-3 Docker run options

Option	Description
faspio-app-sc74cn04	Name of the fasp.io container. We appended the zCX name (sc74cn04) to make it easier to identify the container on each zCX instance.
-p2414:2414/udp	Make available 2414 UDP port.
-p1414:1414	Make available 1414 TCP port.
-d	Run container in background and print container ID.

Option	Description
faspio-img-new	Name of the fasp.io Docker image.

3. Stop the container:

```
docker stop faspio-app-sc74cn04
```

4. Create the gateway.toml file by using the content that is shown in Example 4-8.

Example 4-8 gateway.toml file contents

```
[[bridge]]
  name = "MQZ2_MQZ1"
  [bridge.local]
    protocol = "fasp"
    host = "0.0.0.0"
    port = 2414
  [bridge.forward]
    protocol = "tcp"
    host = "wtsc75.pbm.ihost.com"
    port = 2414
[[bridge]]
  name = "MQZ1_MQZ2"
  [bridge.local]
    protocol = "tcp"
    host = "0.0.0.0"
    port = 1414
  [bridge.forward]
    protocol = "fasp"
    host = "129.40.23.76"
    port = 1414
```

5. Copy the gateway.toml file into the container by running the following command:

```
docker cp gateway.toml faspio-app-sc74cn04:/etc/fasp.io/gateway.toml
```

If you must enable fasp.io in debug mode for troubleshooting, create the logging.toml file by using the content that is shown in Example 4-9.

Example 4-9 Logfile for fasp.io

```
#####
### LOGGING SETTINGS
#####

# Available Loggers:
#
# bridge: High-level logger for the bridge
# s2s: Stream-2-Stream session class logger
# fasp.io-cpp: Logger for the Asio/C++ FASP SDK
# fasp.io-c: Logger for the FASP protocol

# For more on how to configure logging, see full reference at:
https://github.com/guangie88/spdlog\_setup

# level is optional for both sinks and loggers
# level for error logging is 'err', not 'error'
```

```
# _st => single threaded, _mt => multi threaded
# syslog_sink is automatically thread-safe by default, no need for _mt suffix

# check out https://github.com/gabime/spdlog/wiki/3.-Custom-formatting
global_pattern = "%+"

# Async
[global_thread_pool]
queue_size = 8192
num_threads = 1

[[sink]]
name = "console"
type = "color_stdout_sink_mt"

[[logger]]
name = "bridge"
type = "async"
sinks = ["console"]
level = "debug"

[[logger]]
name = "s2s"
type = "async"
sinks = ["console"]
level = "debug"

[[logger]]
name = "fasp.io-cpp"
type = "async"
sinks = ["console"]
level = "debug"

[[logger]]
name = "fasp.io-c"
type = "async"
sinks = ["console"]
level = "debug"
```

6. Copy the logging.toml file into the container by running the following command:

```
docker cp logging.toml faspio-app-sc74cn04:/etc/fasp.io/logging.toml
```

Note: Because the logfile generates a high number of messages, use this file only if it is necessary.

7. Start the new container by running the following command:

```
docker start faspio-app-sc74cn04
```

Complete the following steps to confirm that fasp.io is started:

1. Check the container log to ensure that no issues exist by running the following command:

```
docker logs faspio-app-sc74cn04
```

The expected output is shown in Example 4-10.

Example 4-10 Container log file output

```

Loading gateway config file: /etc/fasp.io/gateway.toml
Loading logging config file: /etc/fasp.io/logging.toml
[2020-06-16 19:22:46.113] [gateway] [info] Gateway version: 1.0.1
5dfd35dde57c7ec251a2fae443f7b05f3e4922b1
[2020-06-16 19:22:46.113] [gateway] [info] Gateway config :
/etc/fasp.io/gateway.toml
[2020-06-16 19:22:46.113] [gateway] [info] Logging config :
/etc/fasp.io/logging.toml
[2020-06-16 19:22:46.116] [gateway] [info] MQZ1_MQZ2: 0.0.0.0:2414 fasp -> tcp
[129.40.23.3]:2414
[2020-06-16 19:22:46.117] [fasp.io-c] [debug] 1:setopt port_reuse:1
[2020-06-16 19:22:46.117] [fasp.io-c] [debug] 1:setopt role:server(1)
[2020-06-16 19:22:46.117] [fasp.io-c] [debug] 1:dgram sock snd buffer requested
10485760 bytes, received 524288 bytes
[2020-06-16 19:22:46.117] [fasp.io-c] [debug] 1:dgram sock rcv buffer requested
10485760 bytes, received 524288 bytes
[2020-06-16 19:22:46.117] [fasp.io-c] [debug] 1:bound [0.0.0.0:2414]
[2020-06-16 19:22:46.117] [fasp.io-c] [debug] 1:detected timer level: 1
[2020-06-16 19:22:46.117] [fasp.io-c] [debug] 1:setopt cipher:none
[2020-06-16 19:22:46.117] [fasp.io-c] [info] 1:FASP Listening [0.0.0.0:2414]
[2020-06-16 19:22:46.117] [gateway] [info] MQZ2_MQZ1: 0.0.0.0:1414 tcp ->
fasp [129.40.23.76]:1414

```

2. Check for any shown container ports by running the following command:

```
docker port faspio-app-sc74cn04
```

The expected output is shown in Example 4-11.

Example 4-11 List-specific mapping for the container

```

1414/tcp -> 0.0.0.0:1414
2414/udp -> 0.0.0.0:2414

```

3. Confirm that fasp.io ports are opened by running the following command:

```
docker exec -it faspio-app-sc74cn04 bash -c "netstat -an | grep 0.0.0.0:*"
```

The expected output is shown in Example 4-12.

Example 4-12 Confirm fasp.io ports are opened

tcp	0	0 0.0.0.0:1414	0.0.0.0:*	LISTEN
udp	0	0 0.0.0.0:2414	0.0.0.0:*	

The fasp.io gateways containers are now running without any problem. You should be able to now integrate with z/OS queue managers.

4.3 Integration with IBM MQ Advanced for z/OS, VUE

This section describes the use of fasp.io with IBM MQ Advanced for z/OS, VUE. The assumption is that the reader has a basic knowledge of IBM MQ Advanced for z/OS, VUE.

4.3.1 Topology

In this section, we build a topology to communicate between two z/OS queue managers by using the Aspera fasp.io gateway. Our topology is shown in the Figure 4-2.

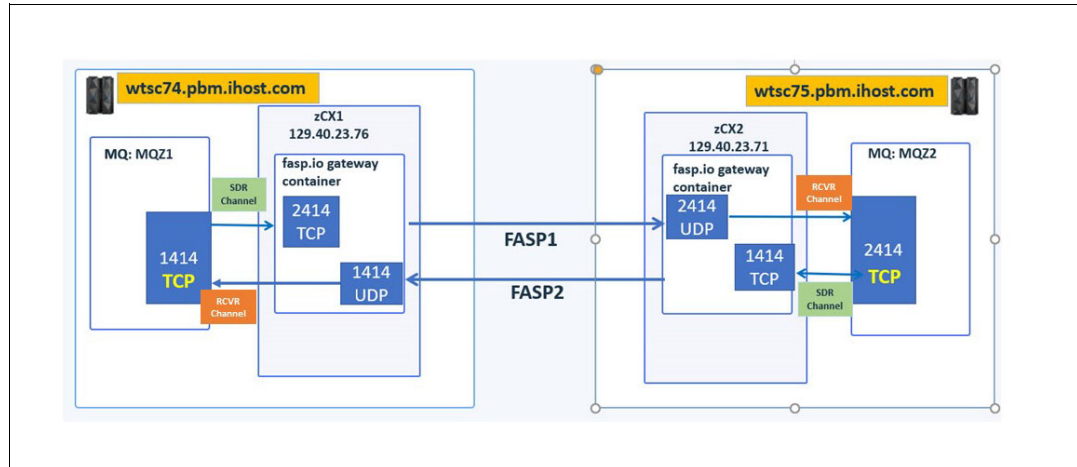


Figure 4-2 IBM MQ and Aspera topology

The topology features the following components:

- ▶ IBM MQ V9.1.5 queue manager on SC74 LPAR: MQZ1
- ▶ IBM MQ listener port on MQZ1: 1414
- ▶ FASP.IO gateway running on zCX instance on SC74:
 - TCP Port: 2414
 - UDP Port: 1414
- ▶ FASP.IO gateway running on zCX instance on SC75
 - TCP Port: 1414
 - UDP Port: 2414
- ▶ IBM MQ V9.1.5 queue manager on SC75 LPAR: MQZ2
- ▶ IBM MQ listener port on MQZ2: 2414

4.3.2 Creating a transmission queue on MQZ1

Run the **MQSC** command to define the transmission queue. We create the transmission queue on queue manager MQZ1 with the name, MQZ2, as shown in Example 4-13.

Example 4-13 Transmission queue on MQZ1

```
DEFINE QLOCAL(MQZ2) USAGE(XMITQ)
```

4.3.3 Creating sender channel on MQZ1

Run the **MQSC** command to define the channel. We create the sender channel on queue manager MQZ1 with the name, MQZ1_MQZ2_0001, as shown in Example 4-14.

Example 4-14 Sender channel definitions on MQZ1

```
DEF CHL(MQZ1_MQZ2_0001) CHLTYPE(SDR) +  
  TRPTYPE(TCP) DISCINT(0) +  
  BATCHSZ(50) BATCHINT(1000) +  
  NPMSPEED(FAST) +  
  XMITQ(MQZ2) +  
  MAXMSGL(2097152) +  
  CONNAME('129.40.23.76(2414)') +  
  MONCHL(QMGR) +  
  COMPMMSG(NONE) +  
  REPLACE
```

4.3.4 Creating receiver channel on MQZ2

Run the **MQSC** command to define the channel. We create the receiver channel on MQZ2, as shown in Example 4-15, with the following name: MQZ1_MQZ2_0001.

Example 4-15 Receiver channel definitions on MQZ2

```
DEFINE CHANNEL(MQZ1_MQZ2_0001) CHLTYPE(RCVR) TRPTYPE(TCP) REPLACE
```

4.3.5 Creating transmission queue on MQZ2

Run the **MQSC** command to define the transmission queue. We create the transmission queue on queue manager MQZ2 with the name, MQZ1, as shown in Example 4-16.

Example 4-16 Transmission queue on MQZ2

```
DEFINE QLOCAL(MQZ1) USAGE(XMITQ)
```

4.3.6 Creating sender channel on MQZ2

We create the sender channel on queue manager MQZ2, as shown in Example 4-17.

Example 4-17 Sender channel definitions on MQZ2

```
DEF CHL(MQZ2_MQZ1_0001) CHLTYPE(SDR) +  
  TRPTYPE(TCP) DISCINT(0) +  
  BATCHSZ(50) BATCHINT(1000) +  
  NPMSPEED(FAST) +  
  XMITQ(MQZ2) +  
  MAXMSGL(2097152) +  
  CONNAME('129.40.23.71(1414)') +  
  MONCHL(QMGR) +  
  COMPMMSG(NONE) +  
  REPLACE
```

4.3.7 Creating receiver channel on MQZ1

Run the **MQSC** command to define the channel. We create the receiver channel on MQZ2 with the following name: MQZ2_MQZ1_0001, as shown in Example 4-18.

Example 4-18 Receiver channel definitions on MQZ2

```
DEFINE CHANNEL(MQZ2_MQZ1_0001) CHLTYPE(RCVR) TRPTYPE(TCP) REPLACE
```

4.3.8 Verifying the infrastructure and configuration

In this section, we create the necessary objects on both queue managers to verify two-way communication.

Creating objects on MQZ1

We created a local queue and a remote queue that are listed in Table 4-4 on queue manager MQZ1.

Table 4-4 IBM MQ Object definitions on MQZ1

Name	Type	Remote Queue	Remote QMGR	XMIT Queue
MQZ1.FASP.MQZ2.QREMOTE	Remote Queue	MQZ1.FASP.MQZ2.QLOCAL	MQZ2	MQZ2
MQZ2.FASP.MQZ1.QLOCAL	Local Queue			

Creating objects on MQZ2

We created a local queue and a remote queue that are listed in Table 4-5 on queue manager MQZ2.

Table 4-5 IBM MQ Object definitions on MQZ2

Name	Type	Remote Queue	Remote QMGR	XMIT Queue
MQZ2.FASP.MQZ1.QREMOTE	Remote Queue	MQZ2.FASP.MQZ1.QLOCAL	MQZ1	MQZ1
MQZ1.FASP.MQZ2.QLOCAL	Local Queue			

4.3.9 Verifying two-way communication

The IBM MQ Explorer tool is used to PUT and GET messages. Complete the following steps:

1. Test the communication from MQZ1 to MQZ2. Open IBM MQ Explorer and select the remote queue, MQZ1.FASP.MQZ2.QREMOTE, as shown in Figure 4-3.

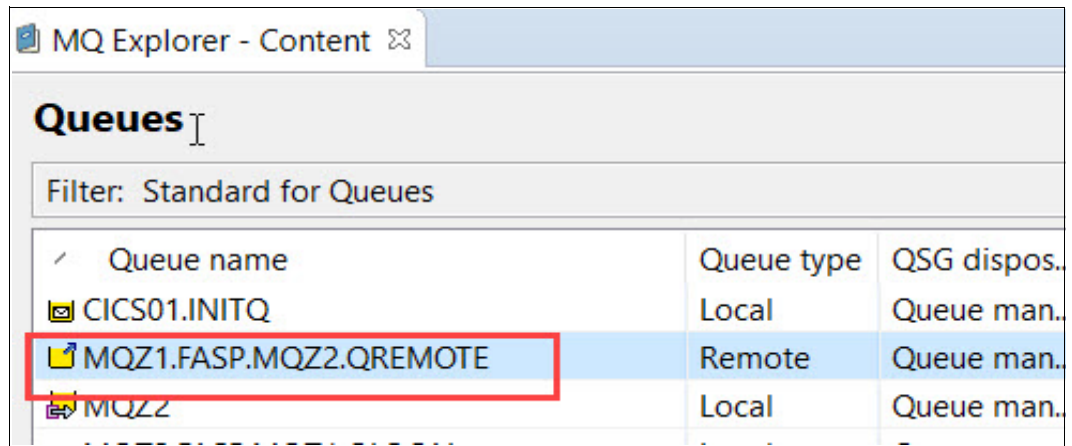


Figure 4-3 Select remote queue on MQZ1

2. PUT a test message on the selected remote queue (see Figure 4-3). Figure 4-4 shows the queue, MQZ1.FASP.MQZ2.QREMOTE.

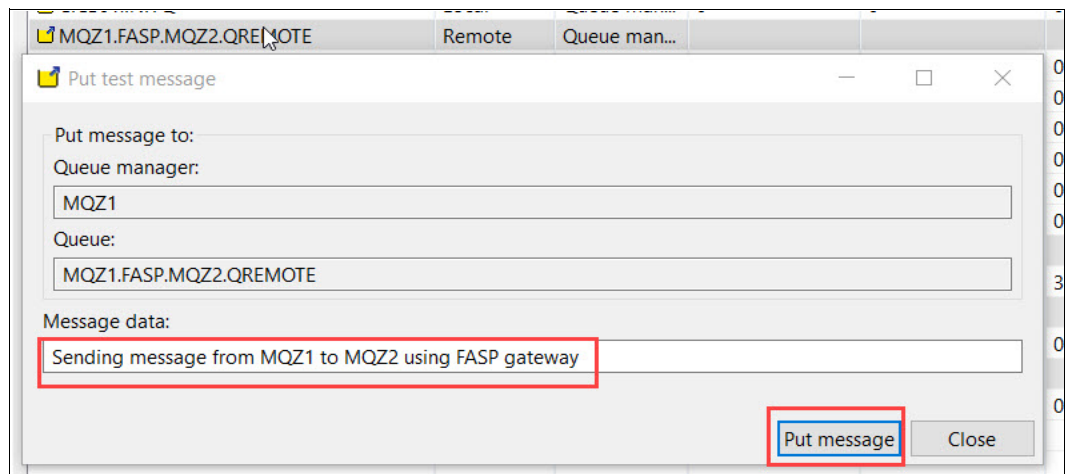


Figure 4-4 Put a test message on MQZ1

3. If our setup is configured correctly, the message should arrive on the local queue that is named MQZ1.FASP.MQZ2.QLOCAL on MQZ2. Select the local queue **MQZ1.FASP.MQZ2.QLOCAL** on MQZ2 and **Browse Messages**, as shown in Figure 4-5.

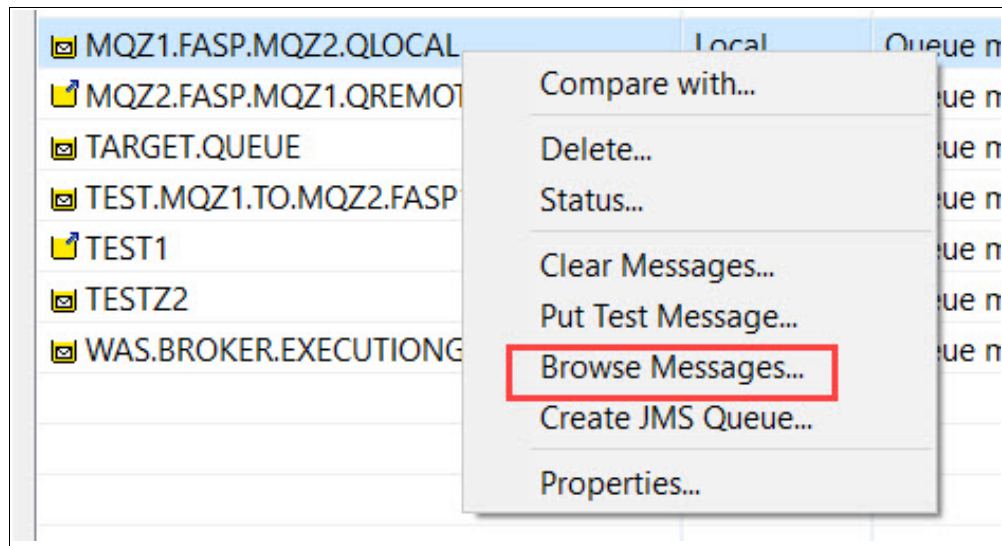


Figure 4-5 Browse messages on MQZ2

We see that the message is delivered to the local queue, as shown in Figure 4-6.

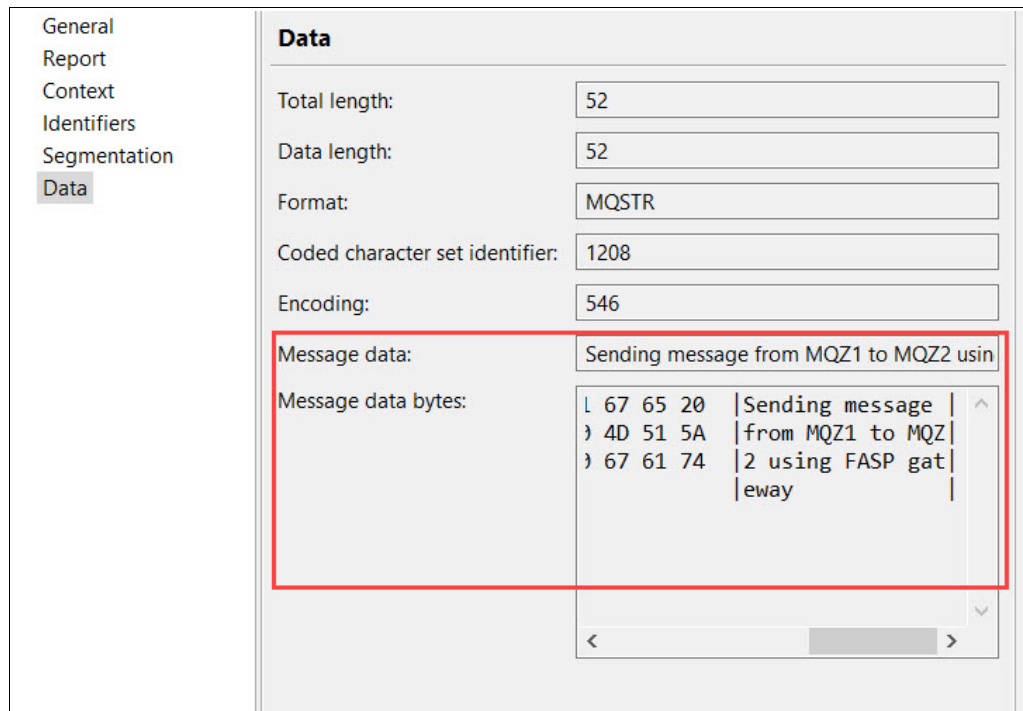


Figure 4-6 Message on MQZ2

This process verifies that our queue managers and FASP.IO gateways are configured correctly for MQZ1 to MQZ2 by using the FASP protocol.

Next, we test the reverse communication from MQZ2 to MQZ1. We PUT a message on the remote queue MQZ2.FASP.MQZ1.QREMOTE on MQZ2.

Browse to the queue manager MQZ1 and Browse Messages on the local queue, MQZ2.FASP.MQZ1.QLLOCAL. We see that a message was delivered to the local queue, as shown in Figure 4-7

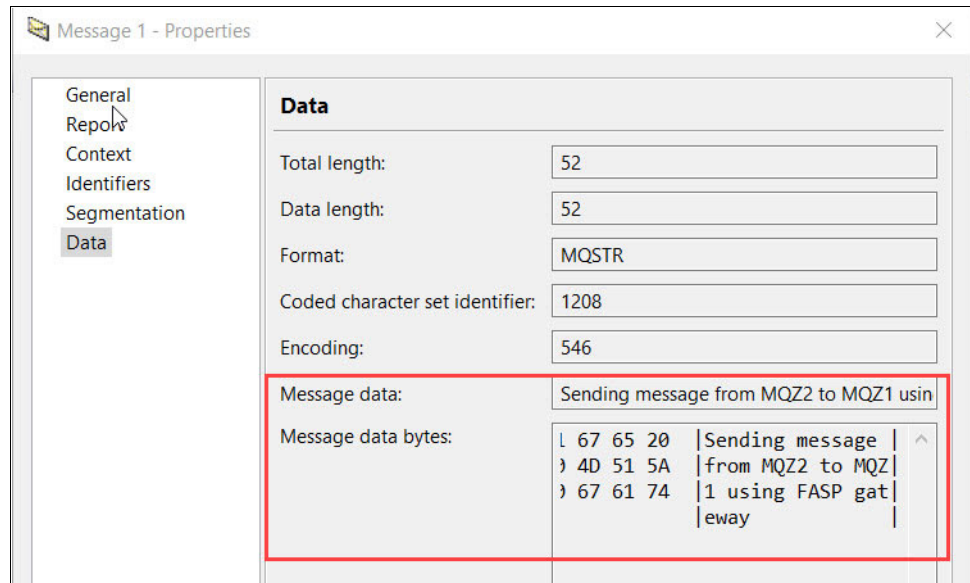


Figure 4-7 Message on MQZ1

It is confirmed that we have two-way communication between the two z/OS queue managers by using the FASP.IO gateway.

4.3.10 Benefits of running the FASP.IO gateway on zCX

By integrating the IBM Aspera FASP protocol, IBM MQ can move messages at high speed over distances for many different real-time processes; for example, financial transactions. The value of having the gateway in zCX is that it makes it as close to the queue manager as possible, which minimizes the amount of time TCP/IP is being used.

Integrating the IBM Aspera FASP protocol also simplifies scenarios where z/OS systems are IPLed in alternative locations, as the FASP.IO gateway function moves automatically with the z/OS system and no changes are needed to the IBM MQ channel configuration to connect to an alternative FASP.IO gateway.



Using IBM MQ on zCX as a client concentrator

zCX brings many benefits, but it is useful if you need to run only a few Linux on IBM Z applications that must connect to your z/OS applications because a dedicated Linux on IBM Z environment or IFLs are not needed. zCX containers can run on zIIPs, which makes them attractive from a pricing perspective.

With zCX comes the ability to make remote connections directly into IBM MQ on the z/OS queue managers. Running the concentrator on the same z/OS LPAR as the target queue manager is efficient and can use the EZAZCX SAMEHOST interface for high-speed communication between the queue managers.

This chapter provides an example use case scenario about how to use IBM MQ on zCX as a client concentrator. It includes the following topics:

- ▶ 5.1, “Interconnecting IBM MQ on z/OS with IBM MQ in zCX” on page 148
- ▶ 5.2, “Using IBM MQ on zCX as a client concentrator” on page 160

5.1 Interconnecting IBM MQ on z/OS with IBM MQ in zCX

This section describes an IBM MQ integration scenario that is based on a messaging infrastructure that uses IBM MQ on z/OS and zCX. We review how to build an IBM MQ container on zCX and create the necessary objects for the integration.

A messaging-based solution establishes a shared integration layer, which enables the seamless flow of multiple types of data between heterogeneous systems.

Many customers have Systems of Engagement (SoE) applications running as cloud native application microservices inside a container. Systems of Record (SoR) applications that are run on z/OS with, for example, CICS, IMS, or DB2 applications. IBM MQ can be used to connect these SoE applications that are running inside zCX with SoR applications that are running on z/OS. SoR applications use a request-reply pattern.

This section explains how a queue manager that is running inside zCX can communicate with a queue manager that is running on z/OS.

Table 5-1 shows how the use of IBM MQ can meet the requirements of SoE and SoR integration.

Table 5-1 SoE and SoR integration requirements

Requirement	Solution
Ability to start CICS/IMS applications from zCX	IBM MQ provides channels that can be created and configured to send messages between applications by using IBM MQ queues.
High availability	IBM MQ can be configured to use a Queue Sharing Group (QSG) on z/OS. Because applications can connect to any queue manager in a QSG, and because all queue managers in a QSG can access shared queues, client applications are not dependent on the availability of a specific queue manager.
High scalability	The use of a QSG enables a scalable solution that takes advantage of the resources across the parallel sysplex. New instances of CICS regions, IMS regions, or queue managers easily can be introduced into the QSG as business growth dictates.
Workload management	Both zCX and IBM MQ can be managed by the WLM.

5.1.1 Architecture

A zCX instance and the IBM MQ queue manager that is run as started tasks on z/OS. CICS and IMS are also started tasks on z/OS. The IBM z/OS Communication Server is used to establish communication between IBM MQ that is running inside the container and IBM MQ that is running on z/OS.

Figure 5-1 shows how applications inside zCX communicate with other z/OS applications by way of the Communication Server.

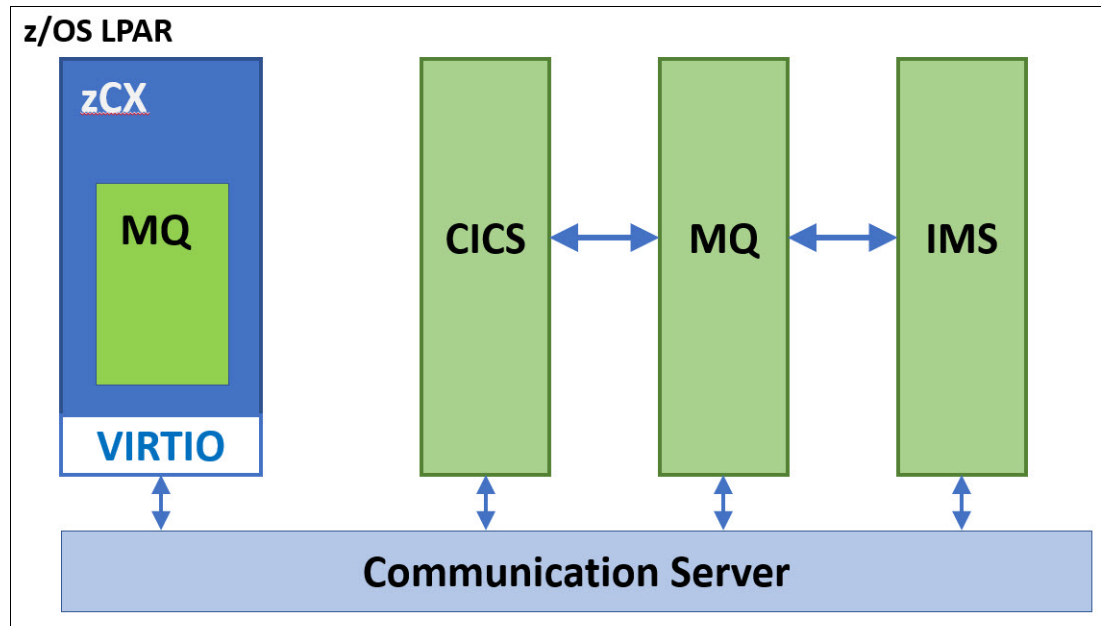


Figure 5-1 Communication between zCX containers and z/OS

5.1.2 Scenario

In this section, we discuss an IBM MQ integration scenario in which two queue managers communicate with each other by way of messages on a queue. For simplicity, we do not build the producer and consumer applications that put and get messages from a queue. CICS and IMS applications can both be producers and consumers of messages.

Cloud native applications running inside a zCX container can also act as consumers or producers. In our example, we use the tools that are provided by IBM MQ to put and get messages from IBM MQ queues.

Figure 5-2 shows the infrastructure to support the communication between the queue managers in our environment.

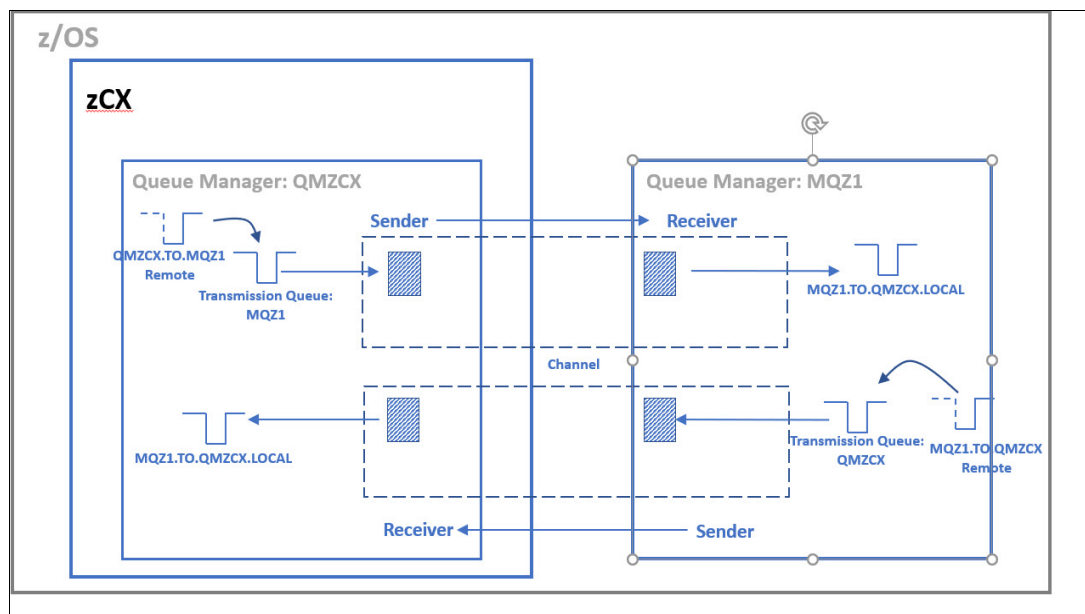


Figure 5-2 Our lab infrastructure

5.1.3 Downloading an IBM MQ image inside zCX from dockerhub

To download the IBM MQ Advanced image from the dockerhub repository, complete the following steps:

1. Browse to the dockerhub IBM MQ Advanced [web page](#).

The display (see Figure 5-3).

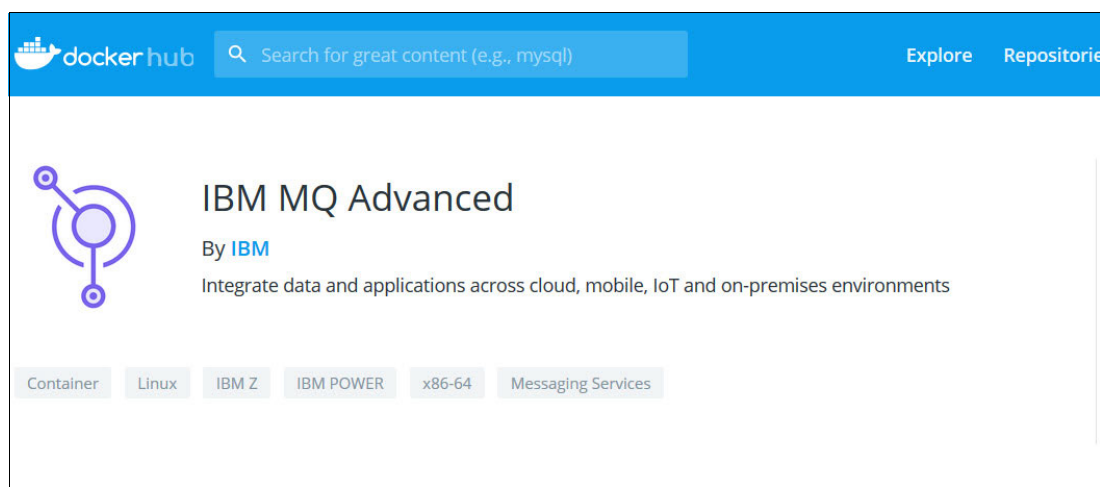


Figure 5-3 IBM MQ Advanced Image on dockerhub

As you scroll through this page, you see listings for maintained and maintained images.

2. Copy the **docker pull** command, as shown in Example 5-1. This example pulls version 9.1.0.0, which is specified as a tag in the command.

Example 5-1 Pull IBM MQ from dockerhub

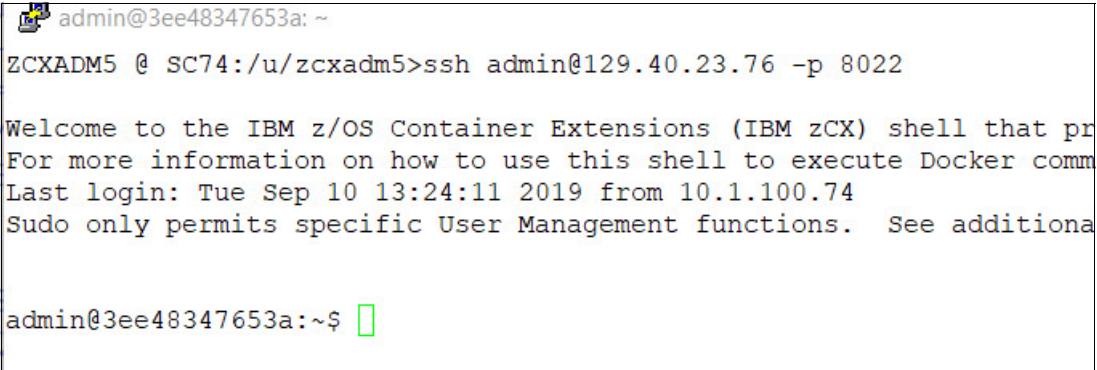
```
docker pull store/ibmcorp/mqadvanced-server-dev:9.1.0.0
```

3. Connect to your previously provisioned zCX instance by way of your SSH client:
 - a. Log on to UNIX System Services by using an SSH client tool, such as PuTTY, with the user ID that you used to create an SSH key. In this example, we created the SSH keys with the ZCXADM5 user ID. Therefore, we logged on with the ZCXADM5 user ID.
 - b. Run the command that is shown in Example 5-2 to connect to the zCX Docker CLI where 129.40.23.76 is the IP address of the zCX instance and 8022 is the listener port.

Example 5-2 Log in as admin to zCX

```
ssh admin@129.40.23.76 -p 8022
```

After successfully logging in, the window looks as shown Figure 5-4.



```
admin@3ee48347653a: ~  
ZCXADM5 @ SC74:/u/zcxadm5>ssh admin@129.40.23.76 -p 8022  
  
Welcome to the IBM z/OS Container Extensions (IBM zCX) shell that pr  
For more information on how to use this shell to execute Docker comm  
Last login: Tue Sep 10 13:24:11 2019 from 10.1.100.74  
Sudo only permits specific User Management functions. See additiona  
  
admin@3ee48347653a:~$
```

Figure 5-4 Successful login

4. Log in to dockerhub with your Docker credentials by running the command that is shown in Example 5-3.

Example 5-3 Log in to docker

```
docker login
```

5. Pull the IBM MQ Advanced image by running the command that is shown in Example 5-4.

Example 5-4 Docker pull command

```
docker pull store/ibmcorp/mqadvanced-server-dev:9.1.0.0
```

Figure 5-5 shows the expected output from logging in and running the `pull` command.

```
admin@570e9473805e: ~
Login Succeeded
admin@570e9473805e:~$ docker pull store/ibmcorp/mqadvanced-server-
9.1.0.0: Pulling from store/ibmcorp/mqadvanced-server-dev
09d85ef3b460: Pull complete
315cc56d0cc9: Pull complete
9ccd8b63257a: Pull complete
beb7e4feb17a: Pull complete
9f9c34f4007e: Pull complete
a0a962a4ce14: Pull complete
b28593d9037e: Pull complete
6c3039323f01: Pull complete
dl5d81c4cc0b: Pull complete
71d9b47f023a: Pull complete
fde8840fd8be: Pull complete
c90f7c3d7c58: Pull complete
0bb8e58d91c5: Pull complete
cc9bf108bd38: Pull complete
1586f0af824f: Pull complete
aeafbc0f2732: Pull complete
bd08b2f08110: Pull complete
11d71aa6c2ae: Pull complete
Digest: sha256:58ff8b97712731a7411cc85d0eb51df322784e827bc6d0116b3
Status: Downloaded newer image for store/ibmcorp/mqadvanced-server
admin@570e9473805e:~$
```

Figure 5-5 IBM MQ Advanced Image Pull

5.1.4 Creating the queue manager inside the container

In this section, we provide the Docker commands that are used to create a Docker volume and the queue manager named QMZCX, which runs as a container inside the zCX instance. Complete the following steps:

1. Create a Docker volume.

A Docker volume must first be created to ensure data persistence. For this example, a volume with the name `mqdataqmcx` is created by running the following command:

```
docker volume create mqdataqmcx
```

The **docker** command, as shown in Example 5-5, creates an IBM MQ Queue Manager named QMZCX that is listening on port 1415 with the Web listener on port 9444 for the IBM MQ Web console. We are also updating to the more current version of IBM MQ, V9.1.2.0, by using a tag for the version.

Example 5-5 Creating Queue Manager on zCX

```
docker run \
  --env LICENSE=accept \
  --env MQ_QMGR_NAME=QMZCX \
  --volume mqdataqmcx:/mnt/mqm \
  --publish 1415:1414 \
  --publish 9444:9443 \
  --detach \
```

store/ibmcorp/mqadvanced-server-dev:9.1.2.0

2. Verify that the Docker container was created and running by running the Docker command that is shown in Example 5-6.

Example 5-6 Display docker process

```
admin@3ee48347653a:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
0c09eade8125	store/ibmcorp/mqadvanced-server-dev:9.1.2.0	
"runmqdevserver"	elastic_bassi	
fa44e995cf5b	store/ibmcorp/mqadvanced-server-dev:9.1.2.0	
"runmqdevserver"	peaceful_bartik	
3ee48347653a	ibm_zcx_zOS_cli_image	"sudo
/usr/sbin/sshd..."	ibm_zcx_zos_cli	

3. Verify that the Queue Manager is created by logging in to the IBM MQ Web Console. Point a web browser to the IBM MQ Web Console with the URL that you used for the provisioned zCX instance, in our case, we used <https://129.40.23.76:9444>. Use the default credentials of username=admin and password=passw0rd to log in to the IBM MQ Web Console. You must change these defaults.

After you log in to the web console, you see a window that is similar to the example that is shown in Figure 5-6.

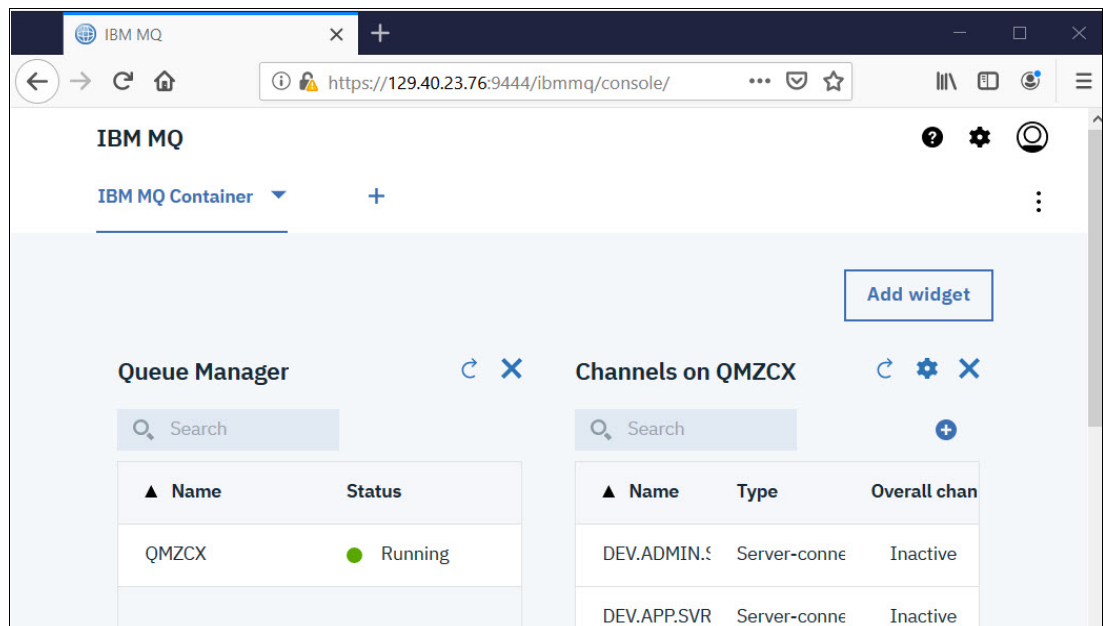


Figure 5-6 IBM MQ Web Console

5.1.5 Creating IBM MQ channels between the queue manager on zCX and z/OS

A channel is a logical communication link. In IBM MQ, two kinds of channels are available:

- Message
- MQI

We create message channels between the queue managers. A message channel connects two queue managers. These channels are unidirectional in that messages are sent only in one direction along the channel. There are subtypes of message channel, with the different types distinguished by whether they are used for receiving or sending messages, whether they start the communication to the partner queue manager, and whether they are used within a cluster.

A channel definition on one queue manager must be matched with a channel definition of the same name with an appropriate subtype on the partner queue manager.

We define a sender channel on one queue manager with an identically named receiver channel on a second queue manager. We also create the appropriate transmission queues for the channels. We do not focus on the IBM MQ commands that are used for creating the channels (assuming that the user knows how to create those channels).

Complete the following steps:

1. Create message channels on QMZCX.

Table 5-2 lists the channels to create on the queue manager named QMZCX.

Table 5-2 Sender-receiver on QMZCX

Channel name	Type	Connection name	Transmission queue
QMZCX.MQZ1	Sender	wtsc74.pbm.ihost.com(1414)	MQZ1
MQZ1.QMZCX	Receiver	NA	NA

Note: The connection named `wtsc74.pbm.ihost.com` is the z/OS LPAR where the queue manager named MQZ1 is running and listening on port 1414. We also created a transmission queue that is a new local queue named MQZ1 with the usage type of XMITQ.

2. Create message channels on z/OS Queue Manager MQZ1.

Table 5-3 lists the channels that are created on queue manager MQZ1 on z/OS.

Table 5-3 Sender-receiver on MQZ1

Channel name	Type	Connection name	Transmission queue
QMZCX.MQZ1	Receiver	NA	NA
MQZ1.QMZCX	Sender	129.40.23.76(1415)	QMZCX

Note: The connection named `129.40.23.76` is the IP address of the provisioned zCX instance. Port 1415 is the port that the queue manager, QMZCX, uses for listening inside the container. We also created a transmission queue that is a new local queue named QMZCX that was created on the queue manager named MQZ1 with the usage type of XMITQ.

This completes the creation of the message channels for queue manager communication.

3. Start sender channels.

Channels often are configured to start automatically when messages are available on the associated transmission queue. We start the sender channels manually to ensure that all the channels are running. We start the sender channels on the queue manager named QMZCX (in the container) and MQZ1 (on z/OS) by completing the following steps:

- a. Log in to the IBM MQ web console, as shown in Figure 5-7. Select the sender channel named **QMZCX.MQZ**. Start the channel and ensure that it is running.

Channels on QMZCX		
Delete	Properties	Start ... 1 item selected
▲ Name	Type	Overall channel s
DEV.ADMIN.SVRCONN	Server-connection	● Inactive
DEV.APP.SVRCONN	Server-connection	● Inactive
MQZ1.QMZCX	Receiver	● Inactive
QMZCX.MQZ1	Sender	● Stopped

Figure 5-7 Channels on QMZCX

- b. We start the sender channel on MQZ1 from the IBM MQ Explorer. Select the sender channel named **MQZ1.QMZCX** and start the channel to ensure that it is running. In our case, the channel did not start and we received an error. To diagnose the error and fix it, complete the following steps:
 - i. On the z/OS Queue Manager, PING the channel MQZ1.QMZCX and look at the results in the channel initiator address space. On the z/OS Console, run the command that is shown in Example 5-7 to ping the channel.

Example 5-7 Ping Channel

```
-MQZ1 ping chl(MQZ1.QMZCX )
```

- ii. The error that is shown in Example 5-8 is returned from the **ping** command. The error indicates that the remote channel is not available. This issue might be the result of the remote queue manager cannot be reached because of security issues.

Example 5-8 Output of Ping Channel

```
-MQZ1 ping chl(MQZ1.QMZCX)
CSQM134I -MQZ1 CSQMPCHL PING CHL(MQZ1.QMZCX) COMMAND ACCEPTED
CSQX558E -MQZ1 CSQXPING Remote channel MQZ1.QMZCX not available
CSQ9023E -MQZ1 CSQXCRPS ' PING CHL' ABNORMAL COMPLETION
```

To inspect the security settings on the QMZCX container queue manager, complete the following steps:

- i. Find the process ID of the container by running the command that is shown in Example 5-9.

Example 5-9 Get process ID of IBM MQ container

```
admin@3ee48347653a:~$ docker ps
```

CONTAINER ID	IMAGE	COMMAND
0c09eade8125	store/ibmcorp/mqadvanced-server-dev:9.1.2.0	"runmqdevserver"
4 hours ago	Up 4 hours	9157/tcp, 0.0.0.0:1415->1414/tcp,
0.0.0.0:9444->9443/tcp	elastic_bassi	
fa44e995cf5b	store/ibmcorp/mqadvanced-server-dev:9.1.2.0	"runmqdevserver"
4 days ago	Up 32 hours	0.0.0.0:1414->1414/tcp,
0.0.0.0:9443->9443/tcp, 9157/tcp	peaceful_bartik	
3ee48347653a	ibm_zcx_zos_cli_image	"sudo
/usr/sbin/sshd..."	5 days ago	Up 33 hours
0.0.0.0:8022->22/tcp		8022/tcp,
	ibm_zcx_zos_cli	

- ii. Run the command that is shown in Example 5-10 to go inside the container to run the IBM MQ commands.

Example 5-10 Docker exec command

```
admin@3ee48347653a:~$ docker exec -it 0c09eade8125 bash
(IBM MQ:9.1.2.0)mqm@0c09eade8125:/$
```

- iii. Display the attributes of queue manager QMZCX by running the **runmqsc** command to run IBM MQ commands on the queue manager and then the MQSC **display qmgr all** command, as shown in Example 5-11.

Example 5-11 Display Queue Manager QMZCX

```
(IBM MQ:9.1.2.0)mqm@0c09eade8125:/$ runmqsc
5724-H72 (C) Copyright IBM Corp. 1994, 2019.
Starting MQSC for queue manager QMZCX.
```

```
dis qmgr all
```

```
1 : dis qmgr all
```

```
AMQ8408I: Display Queue Manager details.
```

QMNAME(QMZCX)	ACCTCONO(DISABLED)
ACCTINT(1800)	ACCTMQI(OFF)
ACCTQ(OFF)	ACTIVREC(MSG)
ACTVCONO(DISABLED)	ACTVTRC(OFF)
ADVCAP(ENABLED)	ALTDATE(2019-09-10)
ALTTIME(18.24.20)	AMQPCAP(NO)
AUTHOREV(DISABLED)	CCSID(819)
CERTLABL(ibmwebspheremqmqzcx)	CERTVPOL(ANY)
CHAD(DISABLED)	CHADEV(DISABLED)
CHADEXIT()	CHLEV(DISABLED)
CHLAUTH(ENABLED)	CLWLDATA()
CLWLEXIT()	CLWLLEN(100)
CLWLMRUC(999999999)	CLWLUSEQ(LOCAL)
CMDEV(DISABLED)	CMDLEVEL(912)
COMMANDQ(SYSTEM.ADMIN.COMMAND.QUEUE)	CONFIGEV(DISABLED)

- iv. We see that **CHLAUTH** is **ENABLED** as a default value and **CONNAUTH** is also set to the default value. For simplicity, we disable **CHLAUTH** and allow **CONNAUTH** to use the ID of the connecting system. We run use the **MQSC ALTER QMGR** command, as shown in Example 5-12.

Example 5-12 Alter queue manager attributes

```
(mq:9.1.2.0)mqm@0c09eade8125:/$ runmqsc
5724-H72 (C) Copyright IBM Corp. 1994, 2019.
Starting MQSC for queue manager QMZX.
```

```
ALTER QMGR CHLAUTH(DISABLED) CONNAUTH(SYSTEM.DEFAULT.AUTHINFO.IDPWOS)
1 : ALTER QMGR CHLAUTH(DISABLED)
CONNAUTH(SYSTEM.DEFAULT.AUTHINFO.IDPWOS)
AMQ8005I: IBM MQ queue manager changed.
```

- v. In IBM MQ Explorer, select the sender channel **MQZ1.QMZX** and start the channel as shown in Figure 5-8. It should go into a running state.

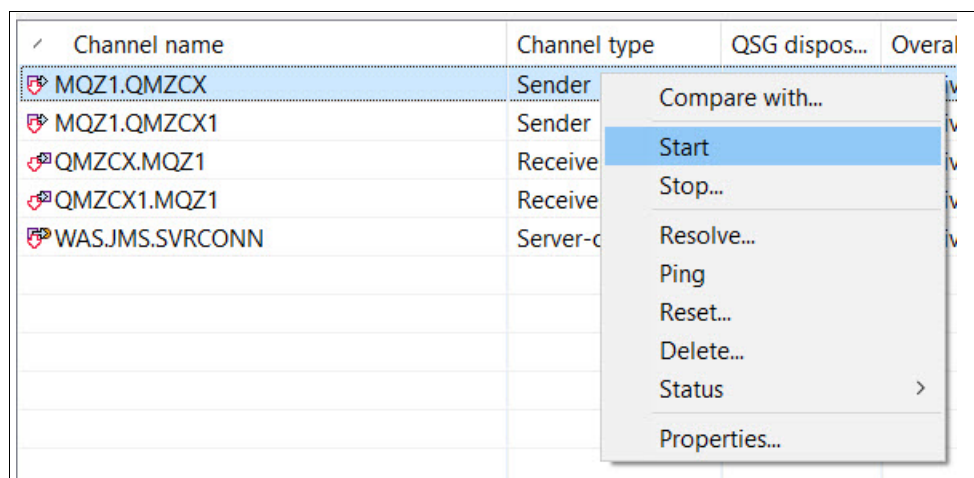


Figure 5-8 Start Sender Channel on MQZ1

5.1.6 Testing the message flow from both directions

We started the sender-receiver channel pairs on both queue managers in the previous step. We now test the infrastructure by putting messages in a queue on one queue manager and receiving the messages on the destination queue.

We created the queues that are listed in Table 5-4 for sending messages from z/OS to zCX.

Table 5-4 From z/OS to zCX

Remote Queue on z/OS (MQZ1)	Local Queue on zCX Container (QMZX)
MQZ1.TO.QMZX.REMOTE	MQZ1.TO.QMZX.LOCAL

We created the queues that are listed in Table 5-5 for sending messages from zCX to z/OS.

Table 5-5 From zCX to z/OS

Remote Queue on zCX Container (QMZCX)	Local Queue on z/OS (QMZ1)
QMZCX.TO.MQZ1.REMOTE	QMZCX.TO.MQZ1.LOCAL

To send messages from z/OS to zCX, complete the following steps.

1. In IBM MQ Explorer, select the remote queue MQZQ.TO.QMZCX and put a test message on the queue as shown in Figure 5-9.

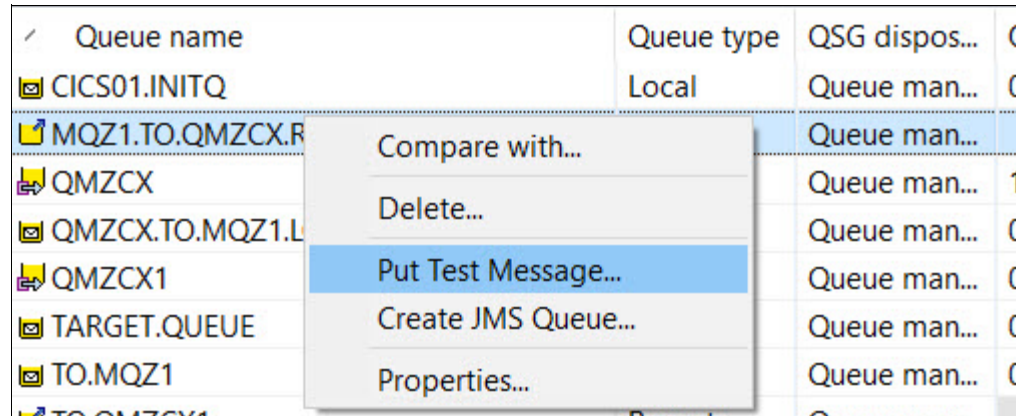


Figure 5-9 Put Test Message on QMZCX1

2. Enter the test message, as shown in Figure 5-10. Click **Put message**.

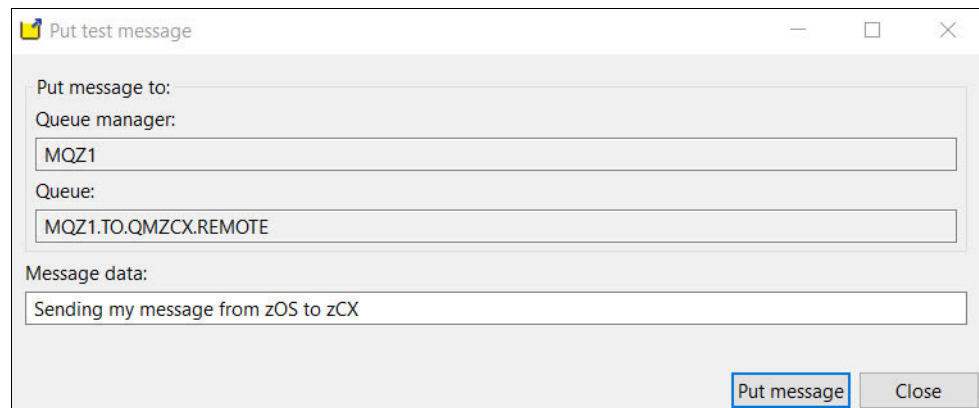
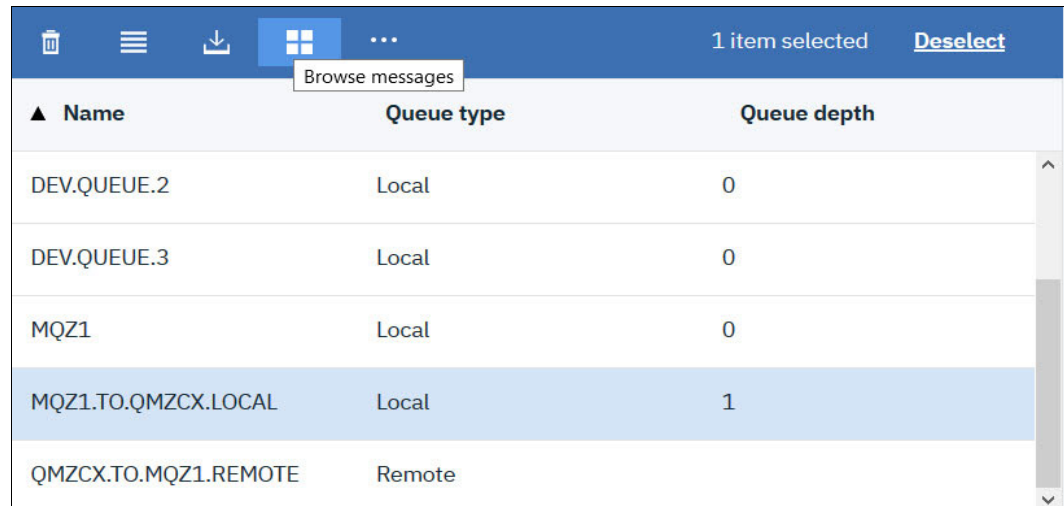


Figure 5-10 Put Message content

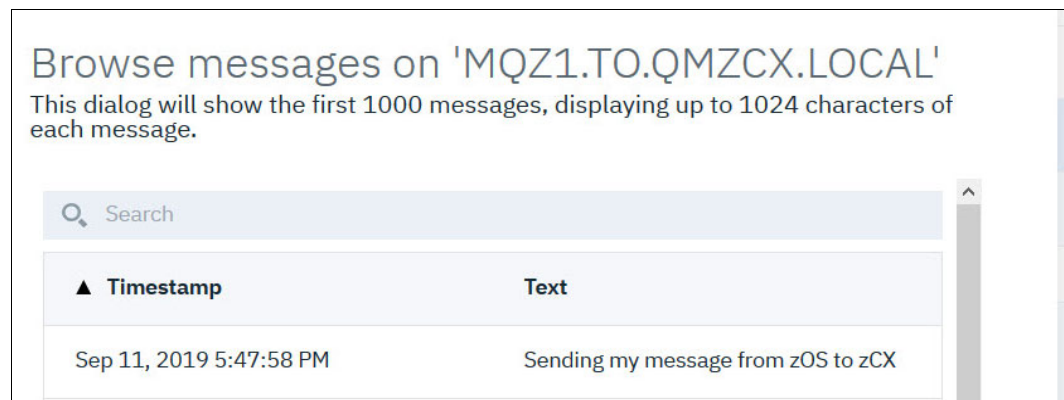
3. Point to the IBM MQ Web Console that is running the IBM MQ container. Select the queue named **MQZ1.TO.QMZCX.LOCAL** and browse the queue, as shown in Figure 5-11.



1 item selected Deselect		
Browse messages		
▲ Name	Queue type	Queue depth
DEV.QUEUE.2	Local	0
DEV.QUEUE.3	Local	0
MQZ1	Local	0
MQZ1.TO.QMZCX.LOCAL	Local	1
QMZCX.TO.MQZ1.REMOTE	Remote	

Figure 5-11 Browse messages on QMZCX

The message browsed is displayed, as shown in Figure 5-12.



Browse messages on 'MQZ1.TO.QMZCX.LOCAL'	
This dialog will show the first 1000 messages, displaying up to 1024 characters of each message.	
<input type="text" value="Search"/>	
▲ Timestamp	Text
Sep 11, 2019 5:47:58 PM	Sending my message from zOS to zCX

Figure 5-12 Content of browsed message

4. Send messages from zCX to z/OS.

Similar steps can be performed by putting a message on the remote queue named QMZCX.TO.MQZ1.REMOTE on QMZCX that is running inside the container. The message can be browsed by using IBM MQ Explorer on the local queue that is named QMZCX.TO.MQZ1.LOCAL on queue manager MQZ1.

5.2 Using IBM MQ on zCX as a client concentrator

We built a queue manager on zCX and setup bidirectional connectivity with a z/OS queue manager. We now describe how this queue manager on zCX can be used in an enterprise.

Large numbers of client applications that are connected directly to an IBM MQ for z/OS queue manager can cause significant CPU load on the channel initiator address space, and therefore increase MLC costs. One way of reducing these costs is to use a client concentrator model where client connections are made to a proxy distributed queue manager (the client concentrator) which then routes messages to and from one or more z/OS queue managers over sender/receiver channels.

Next, we describe how a queue manager running in zCX can be used as a client concentrator.

First, we review the basics of an IBM MQ client.

5.2.1 IBM MQ clients

The IBM MQ client is a lightweight component of IBM MQ that does not require the queue manager runtime code to be on the client system. It enables an application, running on a machine where only the client runtime code is installed, to connect to a queue manager that is running on another machine and perform messaging operations with that queue manager. Such an application is called a *client* and the queue manager is referred to as a *server*.

Using an IBM MQ client is an effective way of implementing IBM MQ messaging and queuing. The following benefits are available by using an IBM MQ client:

- ▶ A licensed IBM WebSphere MQ server installation is not needed on the client machine.
- ▶ Hardware requirements on the client system are reduced.
- ▶ System administration requirements on the client system are reduced.
- ▶ An application that uses an IBM MQ client can connect to multiple queue managers on different machines.

Figure 5-13 shows how IBM MQ clients connect to an IBM MQ server.

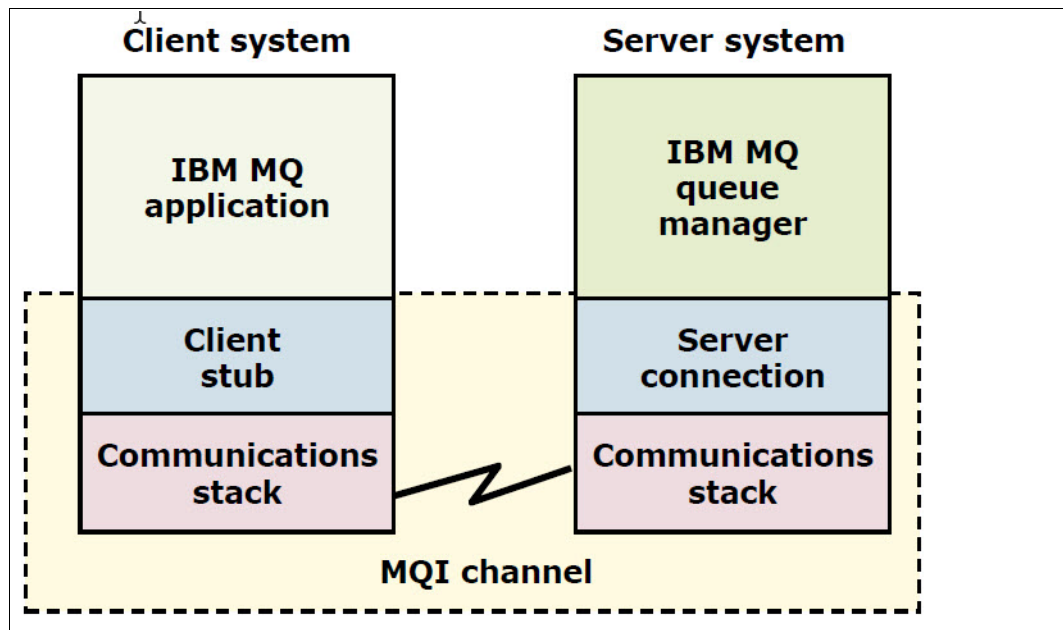


Figure 5-13 MQ Clients

A IBM MQ client includes the following features:

- ▶ IBM MQ client libraries installed
- ▶ No queue managers or queues
- ▶ Uses the same MQI calls as an IBM MQ server
- ▶ Flow of messages in MQI channel is bidirectional

5.2.2 IBM MQ client concentrator

The advantages of the use of a client architecture is that servers are not required to be defined and managed on all of the outlying machines. An enterprise might well have thousands of applications wanting to perform messaging with clients only.

The advantages of the IBM MQ client is that it makes it easier for an increase in IBM MQ client applications to connect to a z/OS queue manager directly. This approach is simple and easy and the topology is similar to the topology that is shown in Figure 5-14.

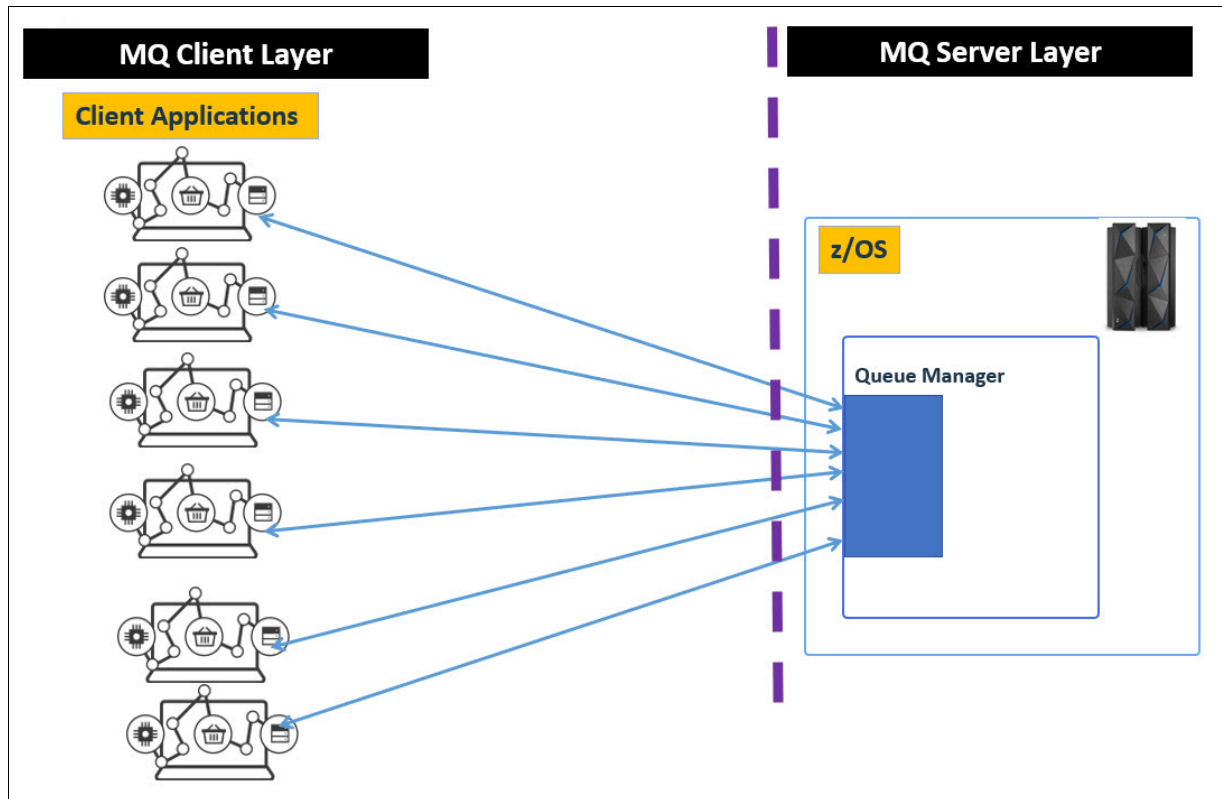


Figure 5-14 Client applications connecting directly

However, direct connection to a z/OS queue manager introduces some challenges. Badly written IBM MQ client applications can perform many connects and disconnects. These frequent connects and disconnects use a large amount of CPU in the **CHINIT** address space.

The CPU overhead on the z/OS address space means more costs. In the client concentrator topology, an extra layer (the client concentrator layer), is introduced, as shown in Figure 5-15.

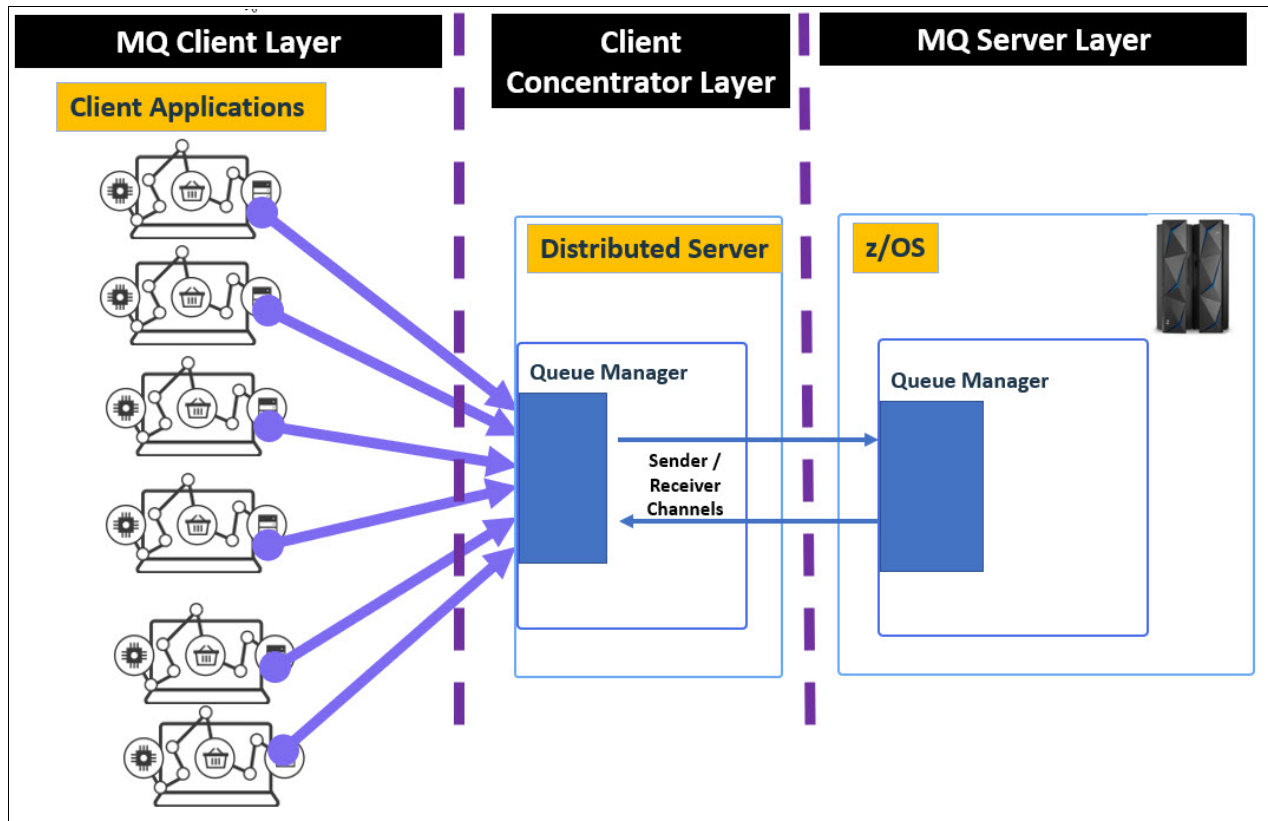


Figure 5-15 Client Concentrator

The high-level architecture of the client concentrator implementation includes the following components:

- ▶ The IBM MQ client layer consists of applications that require IBM MQ connectivity. These applications use the IBM MQ client libraries and are configured to connect to one or more queue managers in the IBM MQ concentrator layer.
- ▶ The client concentrator layer is where all the applications from the IBM MQ client layer connect. This concentrator layer connects to the z/OS queue manager by using sender and receiver channels.
- ▶ The IBM MQ server layer on z/OS provides the messaging platform for the IBM MQ applications

The advantage of using a client concentrator is that it reduces the **CHINIT** CPU usage. The sender and receiver channels are less CPU intensive than the server-connection channels that are used by directly connected client apps.

Note: The concentrator topology is not suitable for all applications. The concentrator layer introduces some latency and complexity and might not be suitable for applications that require high throughput. Also, because it can be a single point of failure, a high availability solution typically is needed.

5.2.3 IBM MQ on zCX as a client concentrator

A queue manager running on zCX can be used as a client concentrator. Figure 5-16 shows the topology of how a queue manager running on zCX can be used as a client concentrator.

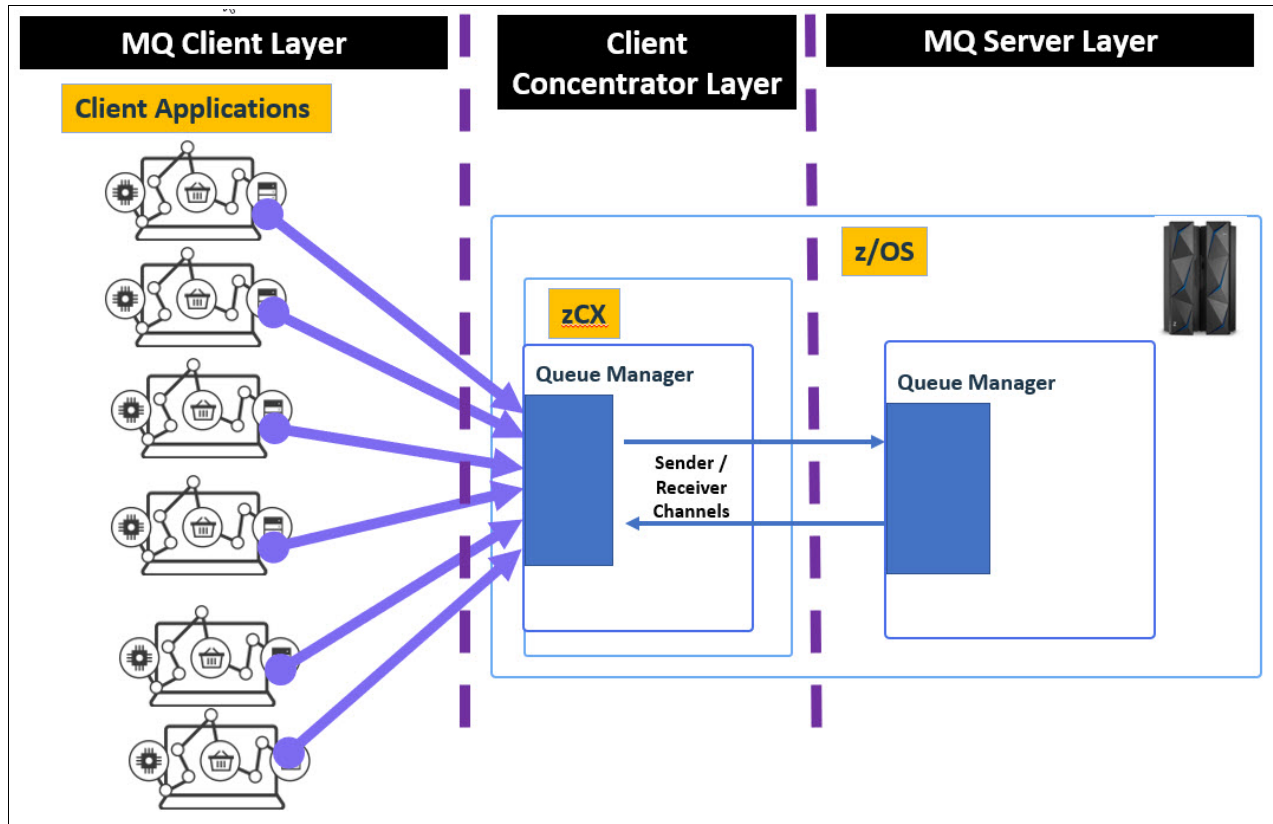


Figure 5-16 IBM MQ on zCX as client concentrator

There are benefits of using this topology, including the following examples:

- ▶ Another distributed server or an IBM MQ Appliance does *not* need to be provisioned.
- ▶ Because the queue manager on zCX and queue manager on z/OS are on the same z/OS LPAR, they use cross memory functionality to implement high-performance network communications.



Part 2

DevOps in zCX

In this part, we describe how organizations can benefit from running a DevOps flow in zCX. We also describe setting up the necessary components.

This section includes the following chapters:

- ▶ “DevOps overview” on page 167
- ▶ “Using Gitea as a code repository” on page 171
- ▶ “Using Jenkins to automate builds” on page 183
- ▶ “Using Ansible to automate deployment and tests” on page 205
- ▶ “Putting it all together and running the pipeline” on page 213

Note: To follow along with the instructions in this part of the IBM Redbooks publication, a running zCX instance and an SSH log on to your zCX instance is required.



DevOps overview

This chapter provides an overview of the DevOps field and describes the benefits of using DevOps within containers and zCX.

This chapter includes the following topics:

- ▶ 6.1, “What is DevOps?” on page 168
- ▶ 6.2, “Why containerize DevOps?” on page 169
- ▶ 6.3, “Why use DevOps in zCX?” on page 169

6.1 What is DevOps?

DevOps is a combination of the words *development* and *operations*. It covers various methods that are based on the principle that understanding the technical and non-technical background is crucial to building a valuable solution.

DevOps is an increasingly common approach that has its roots in agile software development. It enables developers and operations teams to code, build, test, and deploy applications with speed, quality, and control. The core parts of a DevOps flow are shown in Figure 6-1.

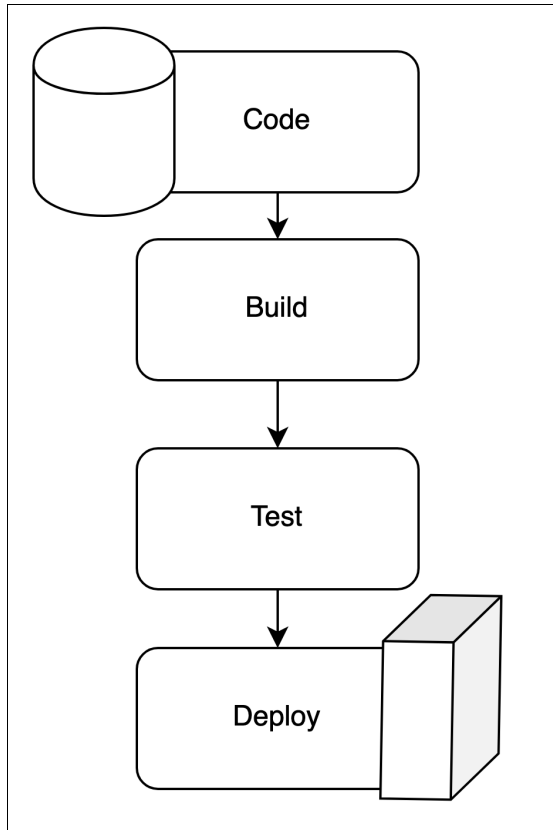


Figure 6-1 Core DevOps components

DevOps puts the interaction between people over processes and tools. The core goal is to achieve a continuous integration and delivery (CI/CD) to create more room for interactions. Successful DevOps implementations generally rely on an integrated set of solutions or a “toolchain” to remove manual repetitive steps, reduce errors, increase team agility, and to scale beyond small, isolated teams.

The use of DevOps is regardless of architecture, platform, or purpose. Common use cases include: cloud-native and mobile applications, application integration, modernization, and multi-cloud management.

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

6.2 Why containerize DevOps?

Although containers are not mandatory for DevOps, they make it much easier and provide some great benefits.

When it comes to resources, containers are best performers. They require fewer system resources than usual machine environments or virtual machines because they do not include operating system images and run directly on the host system.

Containers are real game changers with regards to consistency. While writing, testing, and deploying applications inside containers, the environment does not change at different parts of the CI/CD process. This consistency makes collaboration between different teams easier and brings them closer together because they are all working in the same environment.

In the CI/CD pipeline, it is important to roll out application updates continuously. Containers enable you to easily update your applications. When your application consists of, for example, multiple microservices, each one runs in a separate container and you can deploy your updates step-by-step without stopping the other parts of the application. Therefore, containers allow faster deployment, patching, or scaling of applications.

The use of DevOps comes with the benefit of being independent from specific tools. You can switch between frameworks and platforms easily. Containers are independent from platforms and programming languages and enable you to run nearly every application on every platform. Running your DevOps cycle inside containers ensures that your hosting tools are self-determined, regardless of third parties.

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

6.3 Why use DevOps in zCX?

The use of DevOps within zCX offers the ability to combine powerful container technologies and agile development within the z/OS and IBM Z infrastructure. This feature enables you to take advantage of the benefits from all three worlds: DevOps, containers, and the high performance and security of IBM Z.

In the context of modernizing the mainframe, it is so important to run modern applications on a reliable system. Applications that are hosted in zCX also offer the possibility to dynamically extend your mainframe. An example is a web application that is hosted on zCX that can be a communications entry point for a mobile application.

zCX enables you to bring z/OS enterprise systems, such as CICS and modern applications, to a platform where you can use them together to benefit from their synergies.

Note: Although DevOps features a large cache of tools, only Docker images with s390x architecture can be run in zCX.



Using Gitea as a code repository

Gitea is a self-hosted Git service. It is similar to common services, such as GitHub, Bitbucket, and GitLab. Gitea's goal is to provide an easy and fast way to set up a self-hosted Git service for many platforms and architectures.

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

Note: Using Gitea inside a container offers you independence from third parties and their availability. The use case that is described in this chapter is not dependent on Gitea. You can use any other Git service that provides webhooks. Gitea is our tool of choice in this example because the setup is fast and intuitive.

This chapter describes how to set up Gitea to provide an example of a Git service in zCX and includes the following topics:

- ▶ 7.1, “Setting up Gitea in zCX” on page 172
- ▶ 7.2, “Running a Gitea Docker container” on page 175
- ▶ 7.3, “Checking the Gitea Docker container status” on page 176
- ▶ 7.4, “Configuring Gitea” on page 177

7.1 Setting up Gitea in zCX

This section describes how to build and run a Gitea Docker container and use it as a code repository. We provided code samples and instructions for you to follow along with your own set-up. For more information about instructions for downloadable materials, see Appendix A, “Additional material” on page 253.

7.1.1 Building a Gitea Docker image

Instructions for downloading the files (including the Dockerfile) that are required to build and run a Gitea container for the s390x architecture can be found in the Appendix A, “Additional material” on page 253 of this book. After the download is completed, complete the following steps:

1. Download the added material folder for DevOps and browse from a command prompt to the Gitea folder. Run the following command to copy the file to your z/OS:

```
scp gitea.tar zcxadm3@wtsc74.pbm.ihost.com:/tmp
```

As shown in Example 7-1, you see the process of copying in your console.

Example 7-1 Output of scp gitea.tar command

```
base:Gitea maik$ scp gitea.tar zcxadm3@wtsc74.pbm.ihost.com:/tmp
zcxadm3@wtsc74.pbm.ihost.com's password:
gitea.tar
21% 45MB 135.6KB/s 20:17 ETA
```

2. After completion of the copy process, your file is accessible in z/OS. Log in to your z/OS System by running the following command:

```
ssh -l zcxadm3 wtsc74.pbm.ihost.com
```

3. Browse to the /tmp folder where the gitea.tar file was copied by running the following command:

```
cd /tmp
```

4. To verify that the file was copied correctly, display the files that contain gitea by running the `ls -l | grep gitea` command, which generates the output that is shown in Example 7-2.

Example 7-2 Output of ls -l | grep gitea command

```
ZCXADM3 @ SC74:/SC74/tmp> ls -l | grep gitea
-rw-r--r-- 1 ZCXADM3 SYS1 216485197 Jun 26 12:00 gitea.tar
```

5. To transfer the file securely, log in to your zCX instance by way of SFTP:

```
sftp -P 8022 admin@129.40.23.72
```

Note: In SFTP, the `-P` option is reserved for the port.

You receive the output that is shown in Example 7-3 if the **SFTP** connection was established successfully.

Example 7-3 Output of `sftp -P 8022 admin@129.40.23.72` command

```
ZCXADM3 @ SC74:/SC74/tmp>sftp -P 8022 admin@129.40.23.72
Connected to
129.40.23.72.
```

6. To ensure the right file format when copying files to another system, set file encoding to ASCII by running the **ascii unix** command
7. Copy the `gitea.tar` file from z/OS to your zCX instance by running the **put gitea.tar** command.

As shown in Example 7-4, you see the progress in your console.

Example 7-4 Output of `put gitea.tar` command

```
sftp>put gitea.tar
Uploading gitea.tar to /home/admin/gitea.tar
gitea.tar
100% 206MB 103.2MB/s 00:02
```

8. Now, the `gitea.tar` file is stored in your zCX instance. Run the **exit** command to close the connection.
9. Log in to your zCX instance by running the **ssh admin@129.40.23.72 -p 8022** command and you see the welcome message that is shown in Example 7-5.

Example 7-5 zCX login welcome message

```
ZCXADM3 @ SC74:/u/zcxadm3>ssh admin@129.40.23.72 -p 8022

Welcome to the IBM z/OS Container Extensions (IBM zCX) shell that provides access
to Docker commands.
For more information on how to use this shell to execute Docker commands refer to
IBM
Last login: Fri Jun 26 14:23:27 2020 from 10.1.100.74
Sudo only permits specific User Management functions. See additional
documentation for details.
```

10. You can extract the contents of the `gitea.tar` file by running the **tar -xf gitea.tar** command.
11. Go to the folder where you extracted Gitea by running the **cd gitea** command and verify that the `Dockerfile` is there. Run the **ls -l | grep Dockerfile** command.

You see output that is similar to the output that is shown in Example 7-6.

Example 7-6 Output of `ls -l | grep Dockerfile` command

```
admin@sc74cn05:~$ls -l | grep Dockerfile
-rw-r--r-- 1 admin admin 1298 Jun 24 17:50 Dockerfile
```

12. Because you now have the necessary files for building a Gitea Docker image in your zCX instance, you can run the following build command:

docker build -f Dockerfile -t gitea --build-arg GITEA_VERSION=v1.12.1 .

The output that you receive should contain the Docker image ID and the name tag, as shown in Example 7-7.

Example 7-7 Output of docker build command

```
admin@sc74cn05:~$docker build -f Dockerfile -t gitea --build-arg
GITEA_VERSION=v1.12.1 .
Successfully built 517f0c538739
Successfully tagged gitea:latest
```

7.1.2 Setting up a private registry

If you want to share the image with people who can access your zCX instance or want to have a central place for storing your Docker images securely, you can set up a private registry and push your Docker images to that registry.

Note: Although it is optional to set up a private registry at this point, it is recommended because you need it later in the DevOps flow.

Because the Docker image for a private registry is officially available in Docker Hub for the IBM Z platform, you do not need to build your own image. For more information, see [this web page](#).

Note: If you want to understand how the Dockerfile for the private registry works or do not find a registry image with the version you want to install, you can find the Dockerfile at [this web page](#).

You can copy the file as described for the Gitea Dockerfile in 7.1.1, “Building a Gitea Docker image” on page 172 or you can follow the instructions in the IBM Redbooks publication, *Getting started with z/OS Container Extensions and Docker*, SG24-8457, pages 114 - 116.

To avoid dependencies between the registry container run time and the stored Docker images, it is recommended that you create a volume for storing the registry content outside of the Dockerfile. You create this volume by running the following command:

```
docker volume create registry_data
```

You can run a registry container by referencing the image in the Docker Hub. To do this, run the following command:

```
docker run -d -p 5000:5000 --name registry -v registry_data:/var/lib/registry --rm
ibmcom/registry-s390x:2.6.2.5
```

If the container started successfully, the command returns the container ID. The container ID in our case was:

```
7086f34b2b83a74765a7d04bf40f7ac14be7e0a6532e3584d5b222327035ff2f
```

Because you now have a running registry container, you can tag and push in the Gitea Docker image that you created in 7.1.1, “Building a Gitea Docker image” on page 172:

```
docker tag gitea localhost:5000/gitea
```

Then run:

```
docker push localhost:5000/gitea
```


A Docker image consists of different layers. As you can see in Example 7-8, each layer is pushed to the private registry and receives its own ID.

Example 7-8 Output of docker push localhost:5000/gitea command

```
admin@sc74cn05:~$docker push localhost:5000/gitea
The push refers to repository [localhost:5000/gitea]
c8dfb294bbd4: Pushed
229358ff0258: Pushed
88114cb2d528: Pushed
4752bb9f426e: Pushed
b52e0a82230c: Pushed
d3fde5edeb97: Pushed
latest: digest:
sha256: eaa7c102700dd31a71102625062c5632f3893d879da0bb1813cdbfc03ec9643e size: 1575
```

If the image was pushed successfully into the private registry, you receive the image name and tag, as shown in Example 7-9, by calling its API by running the following curl command:

```
curl 129.40.23.72:5000/v2/gitea/tags/list
```

Example 7-9 Output of curl 129.40.23.72:5000/v2/gitea/tags/list command

```
admin@sc74cn05:~$curl 129.40.23.72:5000/v2/gitea/tags/list
{"name": "gitea", "tags": ["latest"]}
```

7.2 Running a Gitea Docker container

To run a Gitea Docker container, you must first create a volume to ensure the data that is stored in the Gitea container is persistent. To do this, run the following command:

```
docker volume create gitea_data
```

Example 7-10 shows typical output if the creation of the volume was successful.

Example 7-10 Output of Docker volume create gitea_data command

```
admin@sc74cn05:~$docker volume create gitea_data
gitea_data
```

In this section, we describe three options that are available to run a Gitea Docker container:

- Option 1: Run a Gitea Docker container from a Dockerfile

If a Docker image is built in zCX, it can also be run from your local image cache in your instance, as shown in Example 7-11. It is important to map a volume for persistent storage to your container in the run command. Also, it can be useful to add the `--rm` option, which automatically removes the container if it stopped for some reason. This way, it does not block any space in your zCX instance and can be restarted.

Example 7-11 Run a Gitea Docker container command

```
docker run -d -p 3008:3000 -p 24:24 -v gitea_data:/data --name gitea --rm gitea
```

As shown in Example 7-12 the container ID is returned by the run command.

Example 7-12 Output of docker run command

```
admin@sc74cn05:~$docker run -d -p 3008:3000 -p 24:24 -v gitea2:/data --name gitea
--rm gitea
415885ef47f99fd4377546cfa5ef4ffe6539787810997ee074b19c2bfb05169f
```

► Option 2: Run a Gitea Docker container from a private registry

If you store the Gitea image in a private registry, as described in 7.1.2, “Setting up a private registry” on page 174, you can start your Gitea container by using the image name and registry address, as shown in Example 7-13.

Example 7-13 Start the Gitea container

```
docker run -d -p 3008:3000 -p 24:24 -v gitea_data:/data --name gitea --rm
localhost:5000/gitea
```

Again, you see the Container ID as output in the console.

► Option 3: Run a Gitea Docker Container from dockerhub

If you want to start easily with running a Gitea container without building an image, you can pull and run an image that we created from Docker Hub by using the following commands to pull the image and then run it:

```
docker pull maik havemann/gitea-s390x:latest
docker run -d -p 3008:3000 -p 24:24 -v gitea_data:/data --name gitea_data --rm
maik havemann/gitea-s390x
```

The image from the **pull** command is downloaded and copied to your local system. If the **start** command was successful, you see the container ID in the console.

7.3 Checking the Gitea Docker container status

By running the **docker ps** command, you can check whether the container runs and is configured correctly. You should see an output similar to what is shown in Example 7-14.

Example 7-14 Output of docker ps command

```
admin@sc74cn05:~$docker ps
9d7bf05ca731      maik havemann/gitea-s390x
"/usr/bin/entrypoint..."  4 days ago      Up 4 days      22/tcp,
0.0.0.0:24->24/tcp, 0.0.0.0:3008->3000/tcp  gitea
```

7.4 Configuring Gitea

Complete the following steps to configure your running Gitea container:

1. Open the Gitea user interface in a browser. In our case, the IP address is 129.40.23.72:3008.

The user interface that is shown in Figure 7-1 should look the same in your browser.

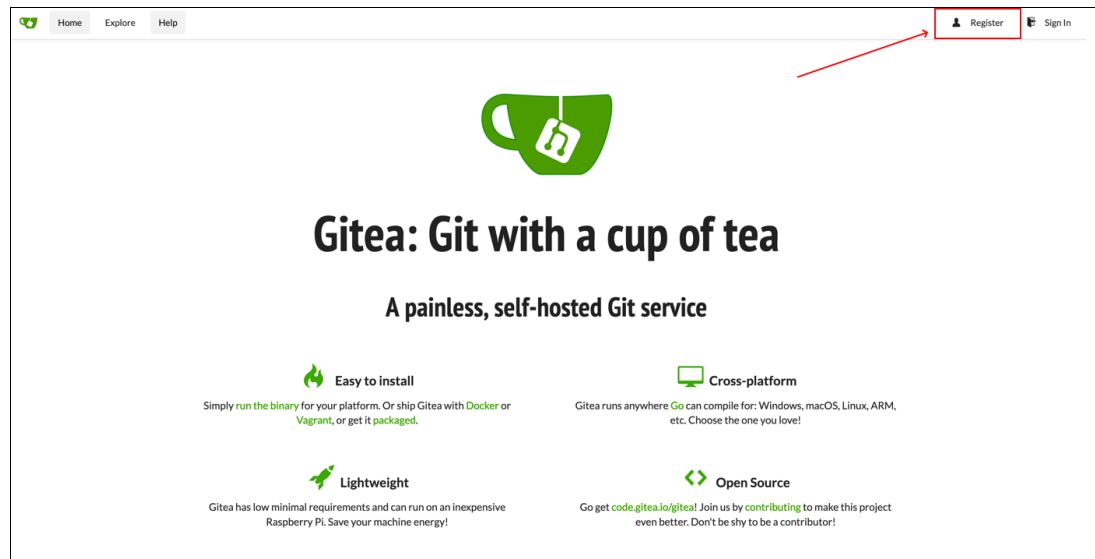


Figure 7-1 Gitea Welcome window

2. Click **Register** in the upper right corner, as shown in Figure 7-1.

3. The Initial Configuration window that is shown in Figure 7-2 opens. Check your Gitea Base URL and HTTP Listen Port and change them if they are not set correctly.

Initial Configuration

If you run Gitea inside Docker, please read the [documentation](#) before changing any settings.

Database Settings

Gitea requires MySQL, PostgreSQL, MSSQL or SQLite3.

Database Type *

SQLite3

Path *

/data/gitea/gitea.db

File path for the SQLite3 database.
Enter an absolute path if you run Gitea as a service.

General Settings

Site Title *

Gitea: Git with a cup of tea

You can enter your company name here.

Repository Root Path *

/data/git/repositories

Remote Git repositories will be saved to this directory.

Git LFS Root Path

/data/git/lfs

Files tracked by Git LFS will be stored in this directory. Leave empty to disable.

Run As Username *

git

Enter the operating system username that Gitea runs as. Note that this user must have access to the repository root path.

SSH Server Domain *

localhost

Domain or host address for SSH clone URLs.

SSH Server Port

22

Port number your SSH server listens on. Leave empty to disable.

Gitea HTTP Listen Port *

3000

Port number the Gitea web server will listen on.

Gitea Base URL *

http://129.40.23.72:3008/

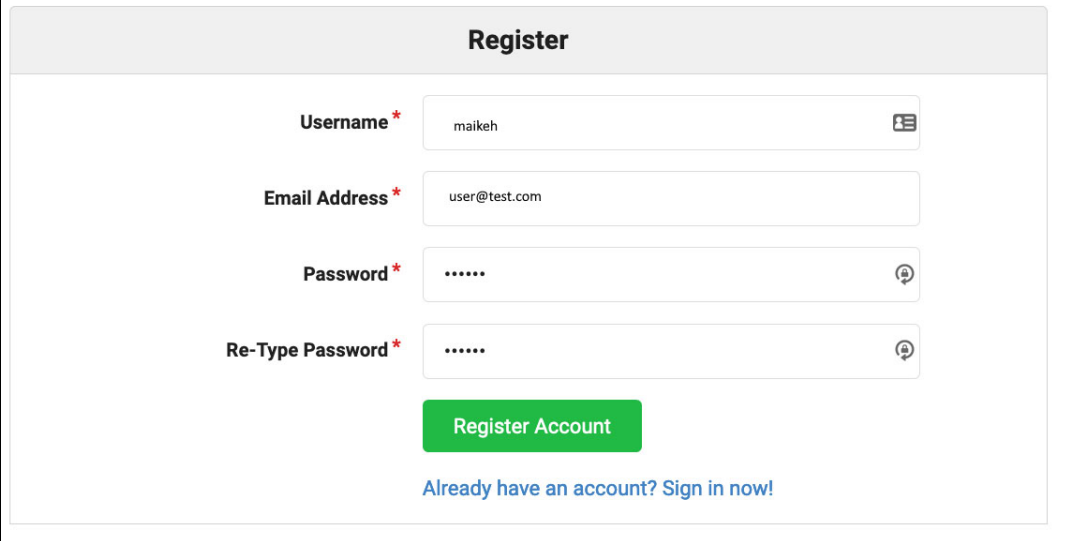
Base address for HTTP(S) clone URLs and email notifications.

Log Path *

/data/gitea/log

Figure 7-2 Gitea Initial Configuration

4. Complete the registration form that is shown in Figure 7-3 to create an account in Gitea.



The image shows the Gitea registration form. It has a title bar 'Register'. Below it are four input fields: 'Username' with the value 'maikeh', 'Email Address' with 'user@test.com', 'Password' with masked characters '.....', and 'Re-Type Password' with '.....'. Each field has a red asterisk indicating it is required. There are icons for password strength and email verification. Below the fields is a green 'Register Account' button and a link 'Already have an account? Sign in now!'.

Figure 7-3 Gitea Register window

7.4.1 Uploading code to Gitea

After the registration, the Gitea overview that is shown in Figure 7-4 appears. To create a code repository, click the plus (+) symbol that is next to “Repositories”.

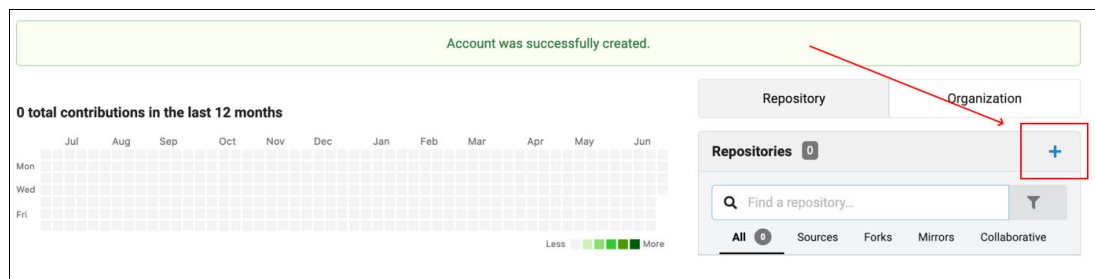




Figure 7-4 Gitea Overview Screen

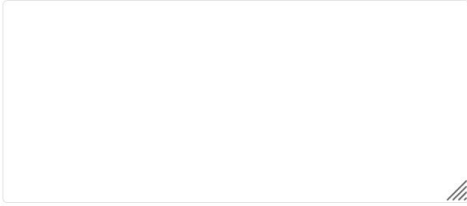
A form is presented for your new repository in which you can name the repository and add a description. Figure 7-5 shows an example of what the form.

New Repository

Owner *  Maikheh
Some organizations may not show up in the dropdown due to a maximum repository count limit

Repository Name * hello-node 
Good repository names use short, memorable and unique keywords.

Visibility ☐ Make Repository Private
Only the owner or the organization members if they have rights, will be able to see it.

Description 

Template Select a template.

Issue Labels Select an issue label set.

.gitignore Select .gitignore templates.

License Select a license file.

README Default

☐ Initialize Repository (Adds .gitignore, License and README)

Default Branch master

Create Repository Cancel

Figure 7-5 Gitea Create Repository

Complete the form and click **Create Repository**.

After a repository is created, you see instructions about how to add code to this repository, as shown in Figure 7-6.



Figure 7-6 Gitea add code by way of command line

In this case, a simple web application that is written in node.js is pushed to the master branch of this repository. It is an application that returns the text Hello zCX User! when called. For more information about the folder that contains instructions about how to download the application files, see Appendix A, “Additional material” on page 253. It is included in the DevOps added material folder.

Note: To deploy the application in zCX, a Dockerfile is required. The repository that is provided in Appendix A, “Additional material” on page 253 contains the necessary Dockerfile.

After pushing code to Gitea, the repository should look similar to the example that is shown in Figure 7-7.

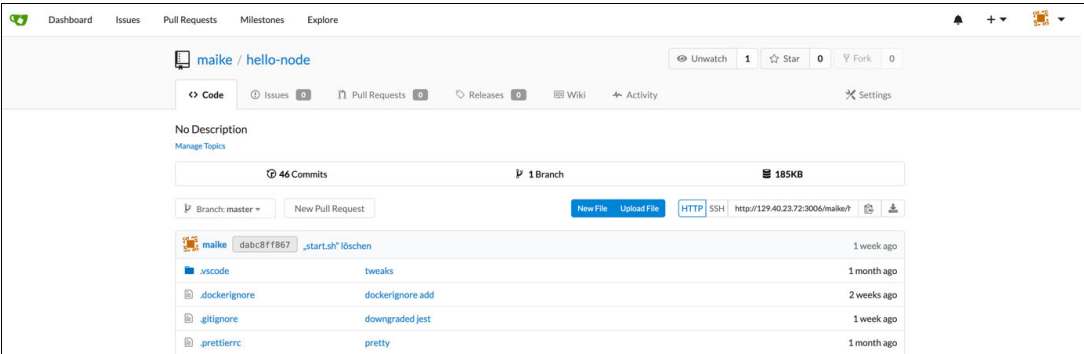


Figure 7-7 Gitea repository hello-node



Using Jenkins to automate builds

This chapter describes how to set up a Docker container for Jenkins server and agent and includes the following topics:

- ▶ 8.1, “What is Jenkins?” on page 184
- ▶ 8.2, “Building a Jenkins server Docker image” on page 184
- ▶ 8.3, “Running a Jenkins server Docker container” on page 187
- ▶ 8.4, “Checking Jenkins server Docker container status” on page 188
- ▶ 8.5, “Configuring Jenkins” on page 188
- ▶ 8.6, “Setting up a Jenkins build agent” on page 196
- ▶ 8.7, “Running a Jenkins build agent Docker container” on page 203
- ▶ 8.8, “Checking Jenkins build agent Docker container status” on page 204

8.1 What is Jenkins?

Jenkins is a self-contained, open source server that can be used to set up a robust CI/CD environment to automate build, test, and deploy tasks. Jenkins can be used within Docker, through native system packages, or as a stand-alone version with a Java Runtime Environment.

After Jenkins is installed, you can install default or custom plug-ins. *Plug-ins* in Jenkins are enhancements of functions to meet specific task requirements. Jenkins offers thousands of plug-ins that can be added manually and installed on the server's user interface.

The Jenkins server or so-called master acts to schedule jobs, assign agents, and send builds to them to run the jobs. The official Jenkins documentation recommends to always set up dedicated build nodes that run separately from the master to free up resources to improve the masters performance.

The Jenkins master uses resources to handle only HTTP requests and manage the environment. Running builds is delegated to the nodes called *agents*. The advantage is that here, builds are prevented from modifying sensitive data in the masters home directory.

The CI/CD pipeline that is run in Jenkins requires a pipeline script. This script defines the build process through the steps of testing and deployment. It is called a Jenkinsfile.

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

8.2 Building a Jenkins server Docker image

The Dockerfile for building a Jenkins server container for the s390x architecture can be found in the Appendix A, "Additional material" on page 253 of this book.

If you have not already done so, download the added material folder for DevOps and browse in the terminal to the folder in which the Dockerfile of the Jenkins server is located.

Run the **scp Dockerfile maikeh@wtsc74.pbm.ihost.com:/tmp** command to copy the file to your z/OS.

As shown in Example 8-1, you see the process of copying in your console.

Example 8-1 Output of scp Dockerfile ... command

```
base:Server files maike$scp Dockerfile maikeh@wtsc74.pbm.ihost.com:/tm
zcxadm3@wtsc74.pbm.ihost.com's password:
Dockerfile
100% 2037      2.0KB/s   00:01
```

After copying the file is accessible in z/OS. Log in to your z/OS system:

```
ssh -l zcxadm3 wtsc74.pbm.ihost.com
```

Then, go to the /tmp folder where the Dockerfile was copied by running the **cd /tmp** command.

To verify that the file was copied correctly, display files that contain Dockerfile by running the **ls -l | grep Dockerfile** command, which generates the output that is shown in Example 8-2.

Example 8-2 Output of ls -l | grep Dockerfile command

```
ZCXADM3 @ SC74:/SC74/tmp> ls -l | grep Dockerfile
-rw-r--r--  1 ZCXADM3 SYS1      2037 Jun 26 13:22 Dockerfile
```

Transfer the file securely by logging in to your zCX instance by using **sftp**:

```
sftp -P 8022 admin@129.40.23.72
```

You receive the output that is shown in Example 8-3 if the **sftp** connection was established successfully.

Example 8-3 Output of sftp -P 8022 admin@129.40.23.72 command

```
ZCXADM3 @ SC74:/SC74/tmp>sftp -P 8022 admin@129.40.23.72
Connected to 129.40.23.72.
```

To ensure the right file format when copying files to another system, set file encoding to ASCII by running the **ascii unix** command.

Then, copy the Dockerfile from z/OS to your zCX instance by running the **put Dockerfile** command.

As shown in Example 8-4, you see the progress in your console.

Example 8-4 Output of put Dockerfile command

```
sftp>put Dockerfile
Uploading Dockerfile to /home/admin/Dockerfile
Dockerfile
100% 2037    2.0KB/s   00:00
```

Now that the Dockerfile file is stored in your zCX instance, and run the **exit** command to close the connection.

The next step is to log in to your zCX instance by running the **ssh admin@129.40.23.72 -p 8022** command and you see the welcome message that is shown in Example 8-5.

Example 8-5 zCX login welcome message

```
ZCXADM3 @ SC74:/u/zcxadm3>ssh admin@129.40.23.72 -p 8022
```

Welcome to the IBM z/OS Container Extensions (IBM zCX) shell that provides access to Docker commands.

For more information on how to use this shell to execute Docker commands refer to IBM

Last login: Fri Jun 26 14:23:27 2020 from 10.1.100.74

Sudo only permits specific User Management functions. See additional documentation for details.

Check to ensure that the Dockerfile is copied successfully into your zCX instance by running the **ls -l | grep Dockerfile** command.

You should see output that is similar to what is shown in Example 8-6.

Example 8-6 Output of `ls -l | grep Dockerfile` command

```
admin@sc74cn05:~$ls -l | grep Dockerfile
-rw-r--r--  1 ZCXADM3 SYS1      2037 Jun 26 13:22 Dockerfile
```

Because you now have the necessary files for building a Jenkins server Docker image in your zCX instance, run the **`docker build -f Dockerfile -t jenkins_server .`** command.

The output that you receive, as shown in Example 8-7, should contain the Docker image ID and the name tag.

Example 8-7 Output of `docker build ... jenkins_server .` command

```
admin@sc74cn05:~$docker build -f Dockerfile -t jenkins_server .
Successfully built 48277d1bcefb
Successfully tagged jenkins_server:latest
```

If you have set up a private registry as described in Chapter 7.1.2, “Setting up a private registry” on page 174, it is now possible to tag and push the Jenkins server Docker image to the registry by running the **`docker tag jenkins_server localhost:5000/jenkins_server`** command.

To push the Docker image to the registry, run the following command:

```
docker push localhost:5000/jenkins_server
```

Example 8-8 shows the output for the push command of the Jenkins server Docker image to the registry.

Example 8-8 Output of `docker push localhost:5000/jenkins_server` command

```
admin@sc74cn05:~$docker push localhost:5000/jenkins_server
The push refers to repository [localhost:5000/jenkins_server]
a883e664bc9b: Pushed
8af47ae9407e: Pushed
efb58ec2d5b2: Pushed
f5f0f8543c72: Pushed
e6592647728f: Pushed
latest: digest:
sha256:46e9bd5458f01393c44da9e6c08b692efb6b708d921fe5629b5b56069c12b98d size: 1365
```

If the image was pushed successfully into the private registry, you see the image name and tag (as shown in Example 8-9) by calling its API by running the following curl command:

```
curl 129.40.23.72:5000/v2/jenkins_server /tags/list
```

Example 8-9 Output of `curl 129.40.23.72:5000/v2/jenkins_server /tags/list` command

```
admin@sc74cn05:~$curl 129.40.23.72:5000/v2/jenkins_server /tags/list
{"name":"jenkins_server","tags":["latest"]}
```

8.3 Running a Jenkins server Docker container

To run a Jenkins server Docker container, you must first create a volume to ensure that the data is persistent in the Jenkins server Docker container. You create the volume by running the **docker volume create jenkins_data** command.

Example 8-10 shows typical output if the creation of a volume was successful.

Example 8-10 Output of docker volume create jenkins_data command

```
admin@sc74cn05:~$docker volume create jenkins_data
jenkins_data
```

In this section, we describe three options that are available to run a Jenkins server Docker container:

- Option 1: Run a Jenkins server Docker container from a Dockerfile

If a Docker image is built in zCX, it can also be run from your local image cache in your instance. It is important to map a volume for persistent storage to your container in the run command. Also, it can be useful to add the **--rm** option, which automatically removes the container if it is stopped for some reason. This way, it does not block any space in your zCX instance and can be restarted.

The default port for the Jenkins user interface is set to 8080. Also, a port 50000 is set for the Jenkins server to receive requests from outside its own container.

To run a Jenkins server Docker container from a Dockerfile, run the following command:

```
docker run -d -p 3000:8080 -p 50000:50000 -v jenkins_data:/data --name
jenkins_server --rm jenkins_server
```

As shown in Example 8-11, the container ID is returned by the **run** command.

Example 8-11 Output of docker run ... jenkins_server command

```
admin@sc74cn05:~$docker run -d -p 3000:8080 -p 50000:50000 -v jenkins_data:/data
--name jenkins_server --rm jenkins_server
5e848255d41ca219524f777a16b0f7a387a57655c207caf1b5407a598f0b4880
```

- Option 2: Run a Jenkins server Docker Container from private registry

If the Jenkins server image was pushed to the private registry, as described in Chapter 8.2, “Building a Jenkins server Docker image” on page 184, you can run the Jenkins server image from the registry by running the following command:

```
docker run -d -p 3000:8080 -p 50000:50000 -v jenkins_data:/data --name
jenkins_server --rm localhost:5000/jenkins_server
```

Again, you see the container ID as output in the console.

- Option 3: Run a Jenkins server Docker Container from dockerhub

If you want to easily start with running only a Jenkins server container without building an image, you can pull and run the image from Docker Hub by running the following command:

```
docker run -d -p 3000:8080 -p 50000:50000 -v jenkins_data:/data --name
jenkins_server --rm maikahavemann/jenkins-server-s390x
```

The image is downloaded and copied to your local system. If the **start** command was successful, the container ID is displayed as output in the console.

8.4 Checking Jenkins server Docker container status

By running the **docker ps** command, you can check whether the container runs and is configured correctly. You should see output similar to what is shown in Example 8-12.

Example 8-12 output of docker ps | grep jenkins_server command

```
admin@sc74cn05:~$docker ps | grep jenkins_server
5e848255d41c      localhost:5000/jenkins_server  "/bin/sh -c 'java -j..."  7
minutes ago      Up 7 minutes      0.0.0.0:50000->50000/tcp,
0.0.0.0:3000->8080/tcp    jenkins_server
```

8.5 Configuring Jenkins

Complete the following steps to configure your running Jenkins server container:

1. Open the user interface in a browser. Browse to the address 129.40.23.72:3000, which leads to the Jenkins Getting Started page, as shown in Figure 8-1.

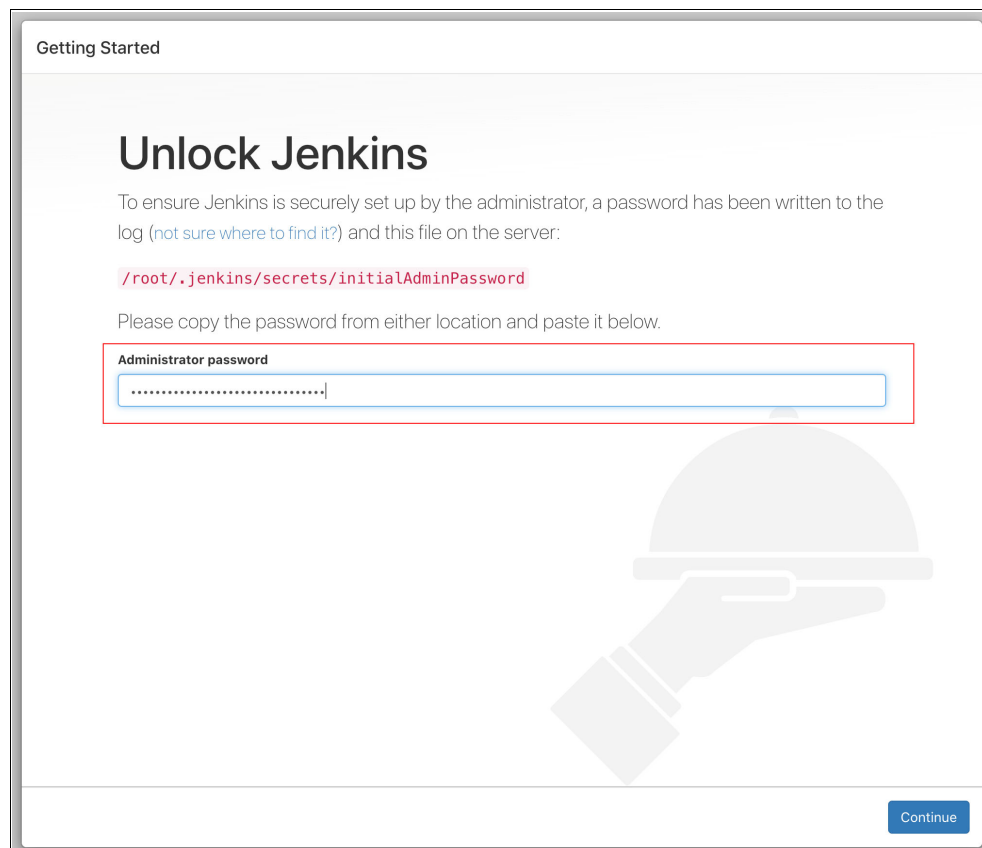


Figure 8-1 Jenkins Start Screen

You get the required initial password by running the `docker logs jenkins_server` command. The password can be found in the command line output, as highlighted in **bold** in Example 8-13.

Example 8-13 Output of docker logs jenkins_server command

```
*****
*****
*****
Jenkins initial setup is required. An admin user has been created and a password
generated.
Please use the following password to proceed to installation:
e7658f1ece7e4dd58ed796c2791e69c1
This may also be found at: /root/.jenkins/secrets/initialAdminPassword
*****
*****
*****
2020-06-26 17:39:18.185+0000 [id=37] INFOjenkins.InitReactorRunner$1#onAttained:
Completed initialization
2020-06-26 17:39:18.217+0000 [id=30] INFOhudson.WebAppMain$3#run: Jenkins is fully
up and running
2020-06-26 17:39:18.380+0000 [id=53] INFOh.m.DownloadService$Downloadable#load:
Obtained the updated data file for hudson.tasks.Maven.MavenInstaller
2020-06-26 17:39:18.384+0000 [id=53] INFOhudson.util.Retrier#start: Performed the
action check updates server successfully at the attempt #1
2020-06-26 17:39:18.387+0000 [id=53] INFO
hudson.model.AsyncPeriodicWork#lambda$doRun$0: Finished Download metadata. 5,423
ms
```

2. Enter the password in the **Administrator password** field of the Jenkins Getting Started window and click **Continue**.

3. In the next window (Figure 8-2), customize Jenkins and install plug-ins as needed.

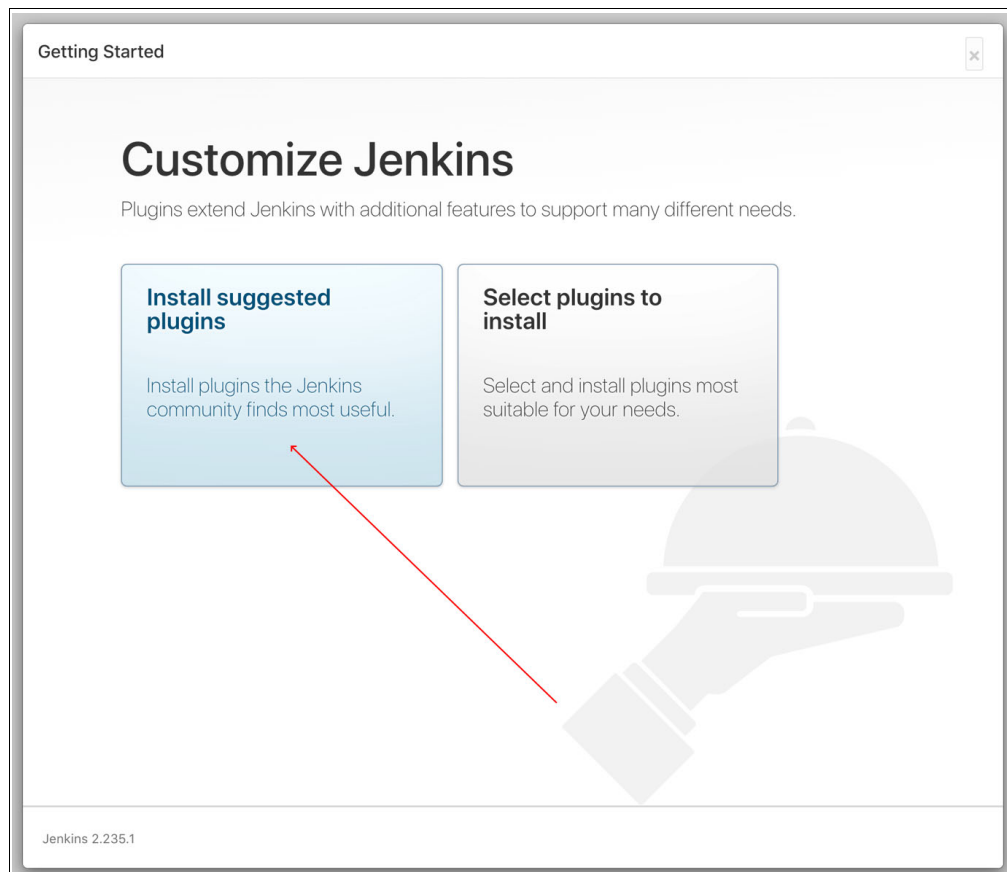


Figure 8-2 Jenkins Install Plug-ins window

4. Install the suggested plug-ins and the progress is displayed in the window, as shown in Figure 8-3.

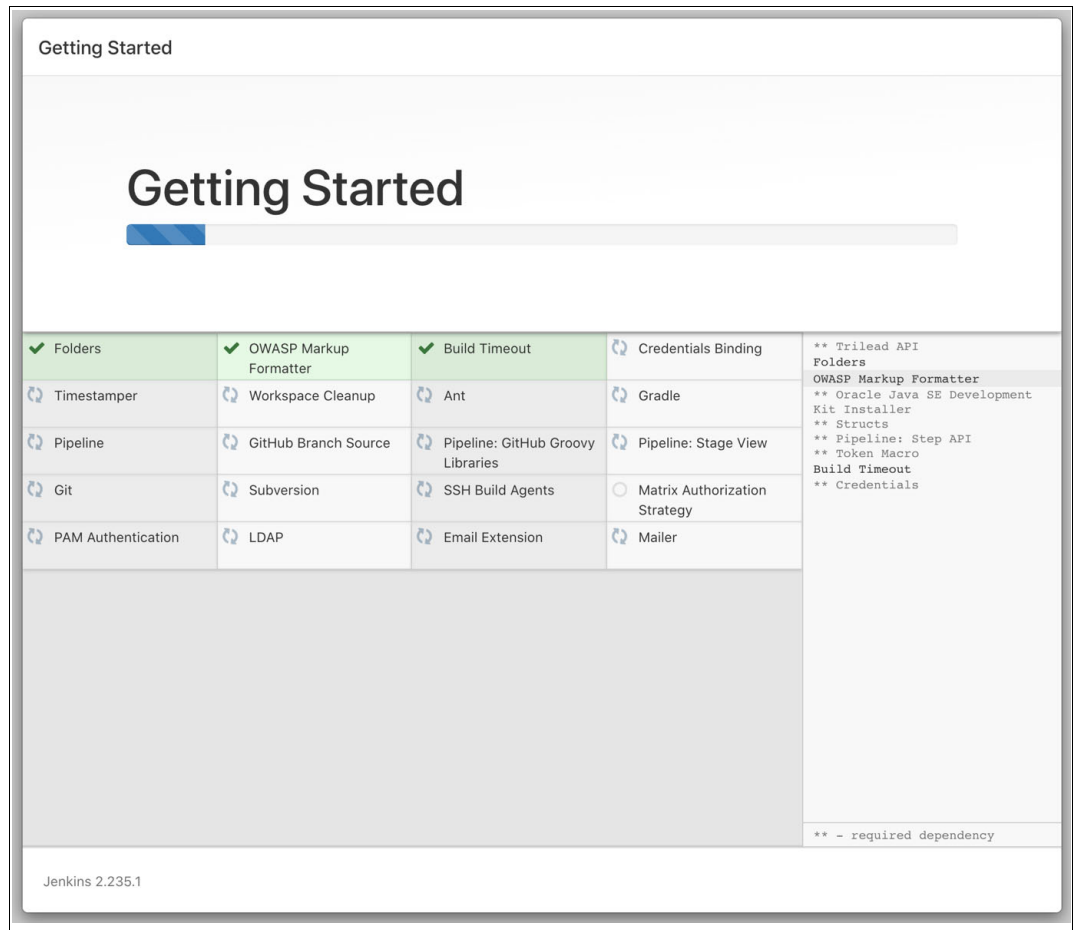
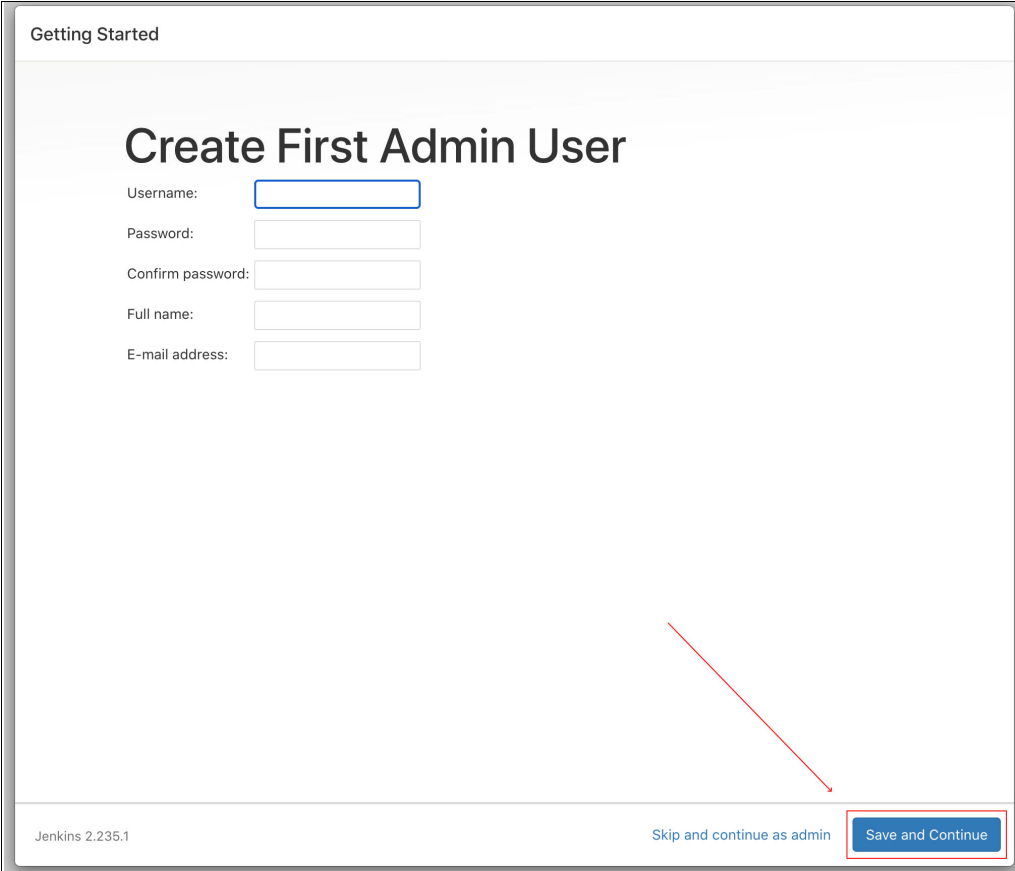


Figure 8-3 Jenkins Progress Install Plug-ins window

Note: Occasionally, the installation of some plug-ins might fail. Plug-ins can be installed later manually. This process is demonstrated by using the Ansible plug-in that is described in 8.5.1, “Installing plug-ins manually” on page 194.

5. Create an administrative user for your Jenkins server, as shown in Figure 8-4.



The screenshot shows the 'Getting Started' page of Jenkins 2.235.1. The main heading is 'Create First Admin User'. Below it are five input fields: 'Username:', 'Password:', 'Confirm password:', 'Full name:', and 'E-mail address:'. The 'Username' field is highlighted with a blue border. At the bottom right, there are two buttons: 'Skip and continue as admin' and 'Save and Continue'. The 'Save and Continue' button is highlighted with a red border, and a red arrow points to it from the right side of the form area.

Getting Started

Create First Admin User

Username:

Password:

Confirm password:

Full name:

E-mail address:

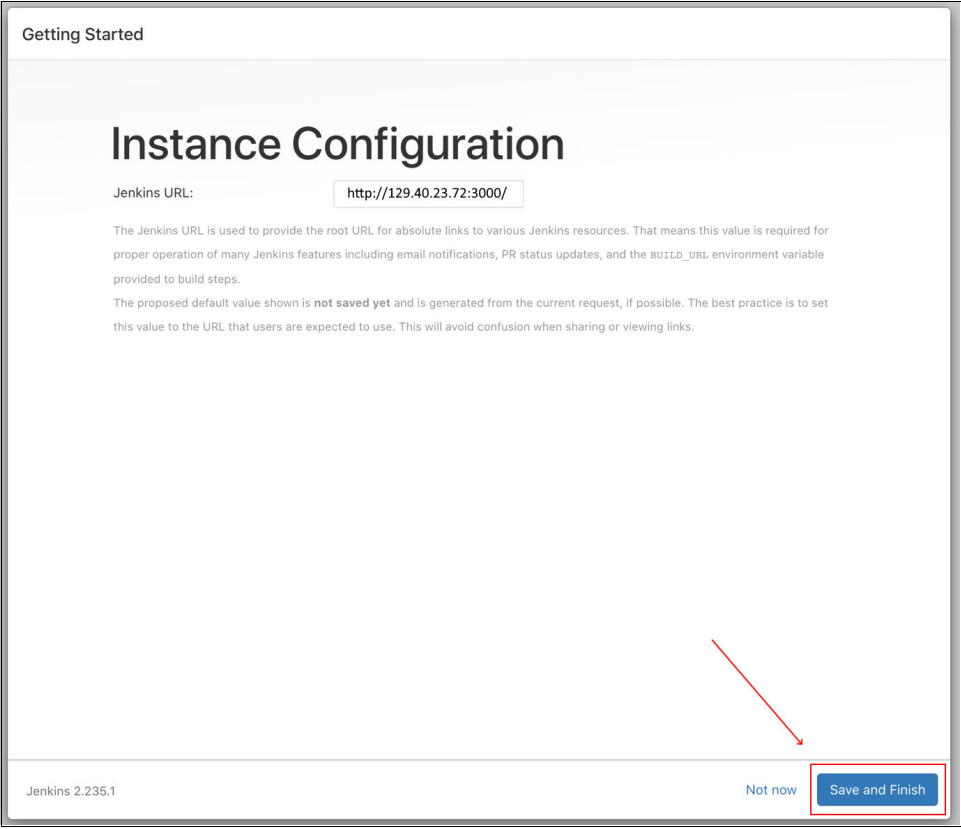
Jenkins 2.235.1

[Skip and continue as admin](#)

Figure 8-4 Jenkins Create Admin User

6. After entering your credentials, click **Save and Continue**.

7. As shown in Figure 8-5, enter your Jenkins server URL into the configuration.



Getting Started

Instance Configuration

Jenkins URL:

The Jenkins URL is used to provide the root URL for absolute links to various Jenkins resources. That means this value is required for proper operation of many Jenkins features including email notifications, PR status updates, and the `BUILD_URL` environment variable provided to build steps.

The proposed default value shown is **not saved yet** and is generated from the current request, if possible. The best practice is to set this value to the URL that users are expected to use. This will avoid confusion when sharing or viewing links.

Jenkins 2.235.1

[Not now](#) [Save and Finish](#)

Figure 8-5 Jenkins Instance Configuration

8. Click **Save and Finish** and the “Jenkins is ready!” message is displayed, as shown in Figure 8-6.

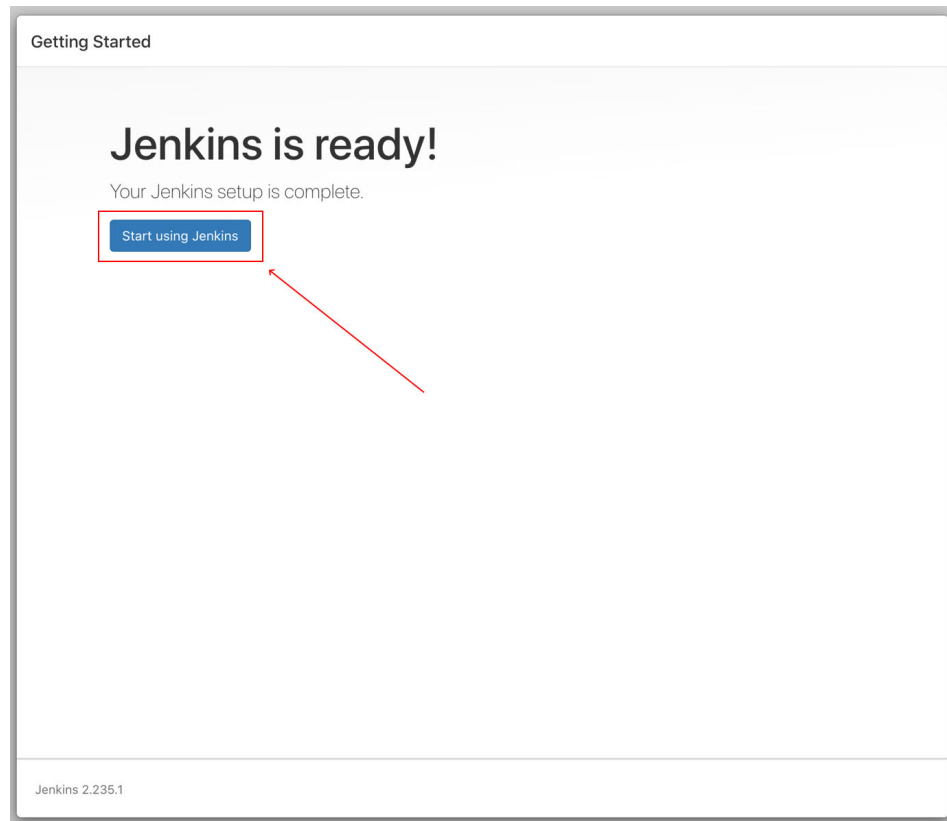


Figure 8-6 Jenkins Is Ready

The initial Jenkins setup is complete. Click **Start using Jenkins** (see Figure 8-6).

8.5.1 Installing plug-ins manually

The Ansible plug-in is already installed. Complete the following steps to install a plug-in manually. Ansible is required later on for use in the application deployment in 9.2, “Setting up Ansible by using Jenkins” on page 206:

1. When the Jenkins setup is complete, you see the Jenkins home window that is shown in Figure 8-7. To install a plug-in manually, click **Manage Jenkins**.

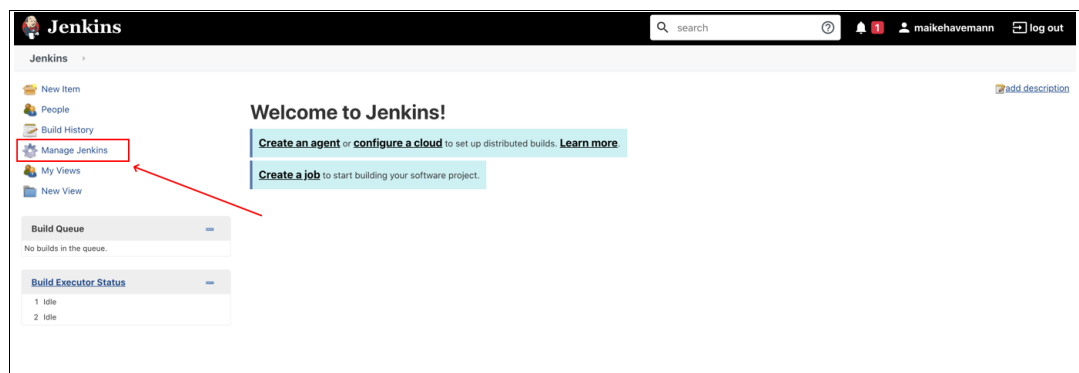


Figure 8-7 Jenkins Home window

2. Click **Manage Plugins**, as shown in Figure 8-8.

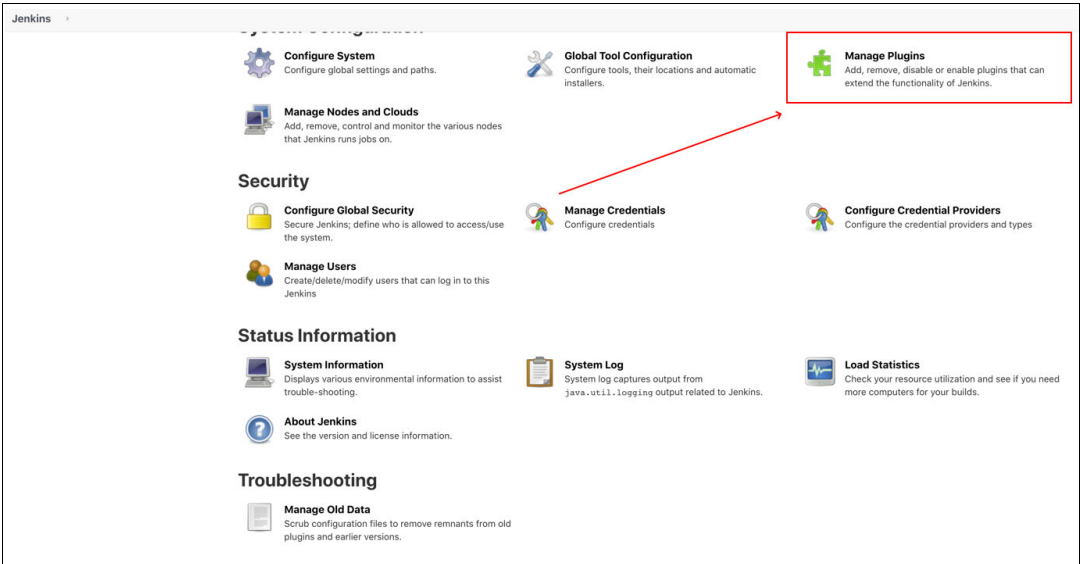


Figure 8-8 Manage Jenkins menu

3. Search for the Ansible plug-in, select the box, and click **Download and install after restart** (see Figure 8-9).

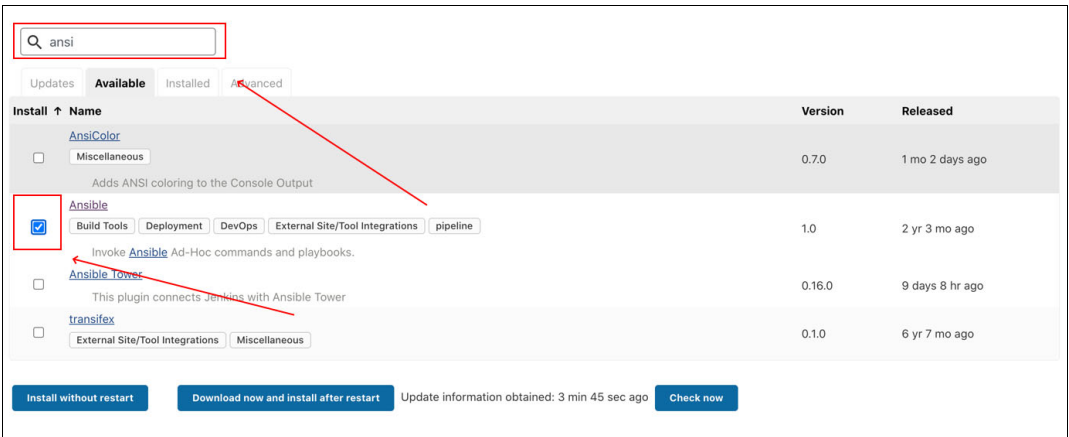


Figure 8-9 Install Ansible

4. After the installation is complete, click **Restart Jenkins**. The restart window is displayed, as shown in Figure 8-10.

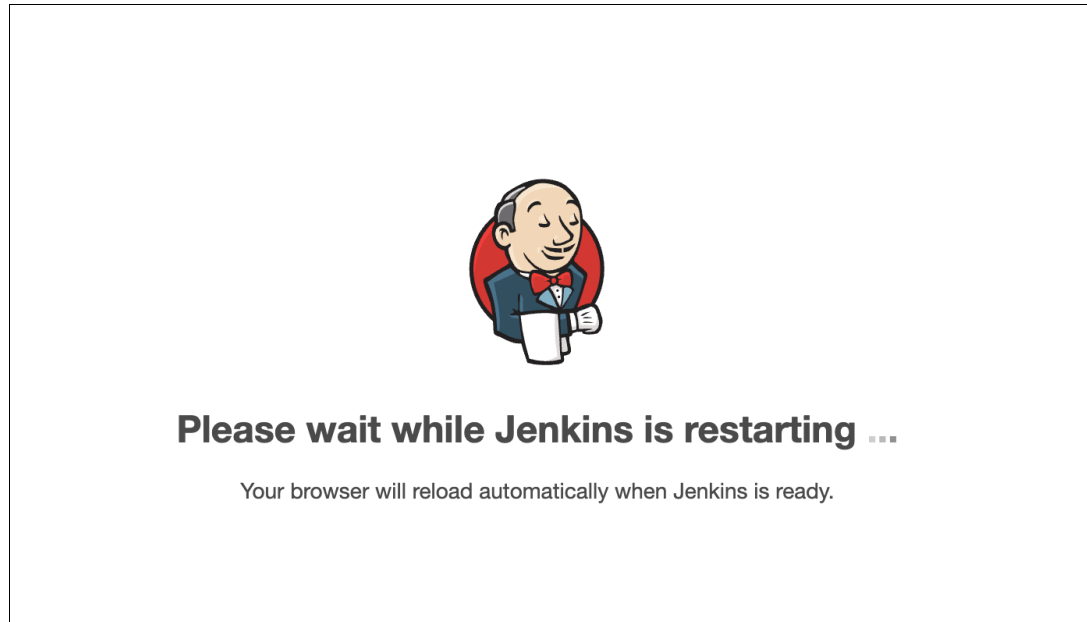


Figure 8-10 Jenkins Restart window

8.6 Setting up a Jenkins build agent

The purpose of Jenkins is to automate build jobs. Jobs are run by dedicated agents. To build the image of the node.js example application, an agent is set up.

In this section, we describe how to create a Jenkins build node and build the Jenkins agent Docker image. We also provide two options to run a Jenkins build agent Docker container.

8.6.1 Creating a Jenkins build node

Complete the following steps to create a Jenkins build node:

1. From the Jenkins Home window (see Figure 8-7), click **Manage Jenkins**. As shown in Figure 8-11, select **Manage nodes and clouds**.

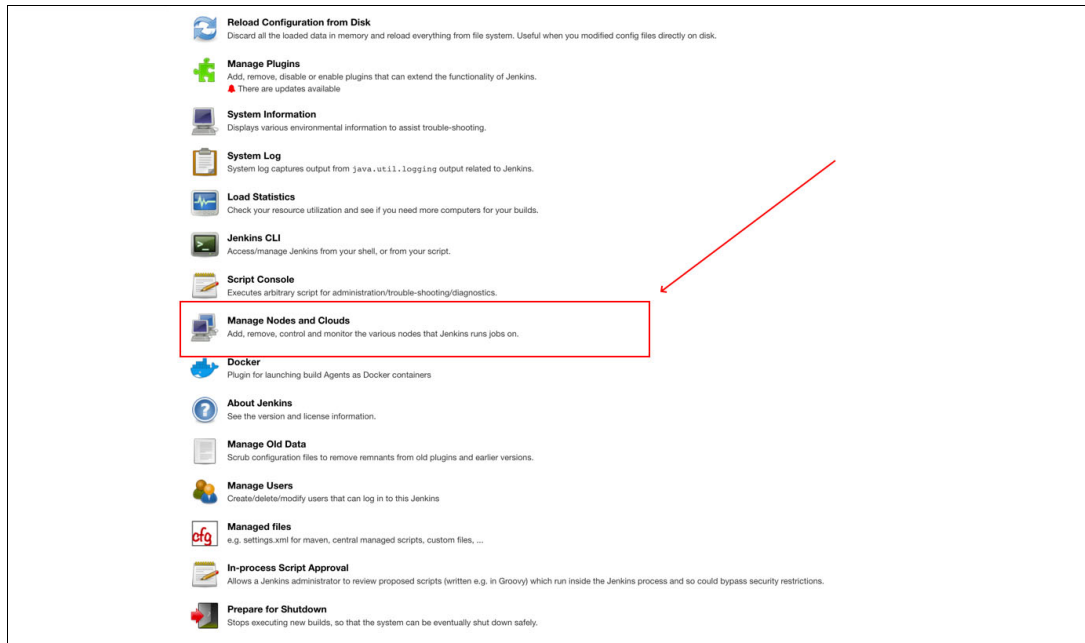


Figure 8-11 Jenkins Manage Nodes

2. Click **New Node**, as shown in Figure 8-12.

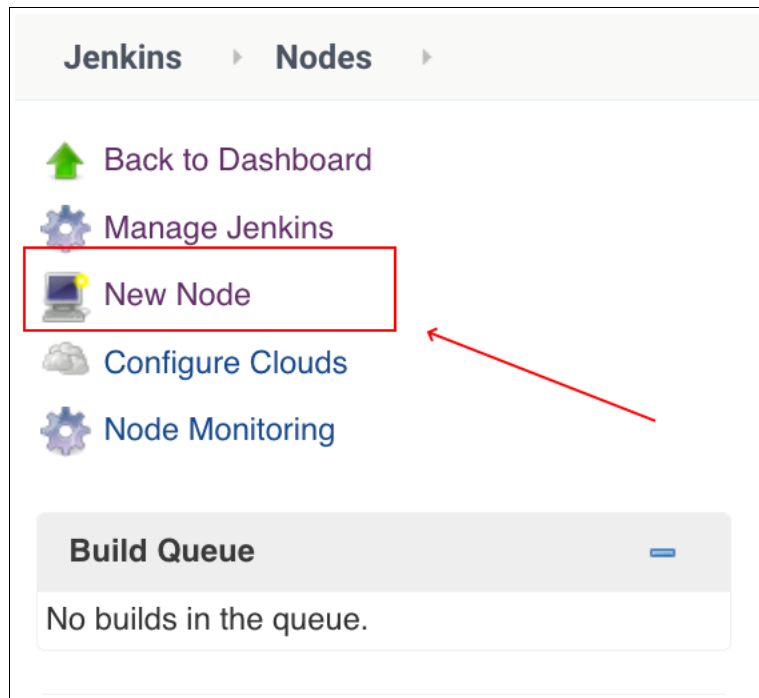


Figure 8-12 Selecting New Node option

3. Create the agent as a permanent agent and configure it as shown in Figure 8-13.

Figure 8-13 Configure Build Node

4. You are provided with an overview of your agent. Copy the agent secret that is highlighted in Figure 8-14 (you need the secret later to run the build agent container).

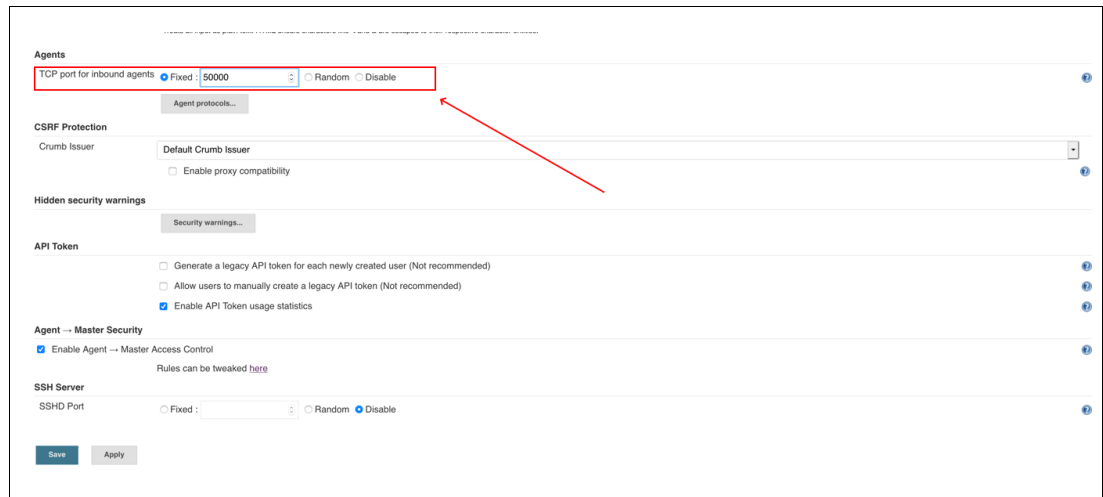
Figure 8-14 Jenkins Build Agent

To make sure that the agent container can communicate with the Jenkins server, the server's TCP inbound port must be set to 50000. This port was set in 8.3, "Running a Jenkins server Docker container" as a communication endpoint for the Jenkins server.

5. Return to the Manage Jenkins window (see Figure 8-7). Then, select the Jenkins security menu, as shown in Figure 8-15.

Figure 8-15 Jenkins Configure Global Security

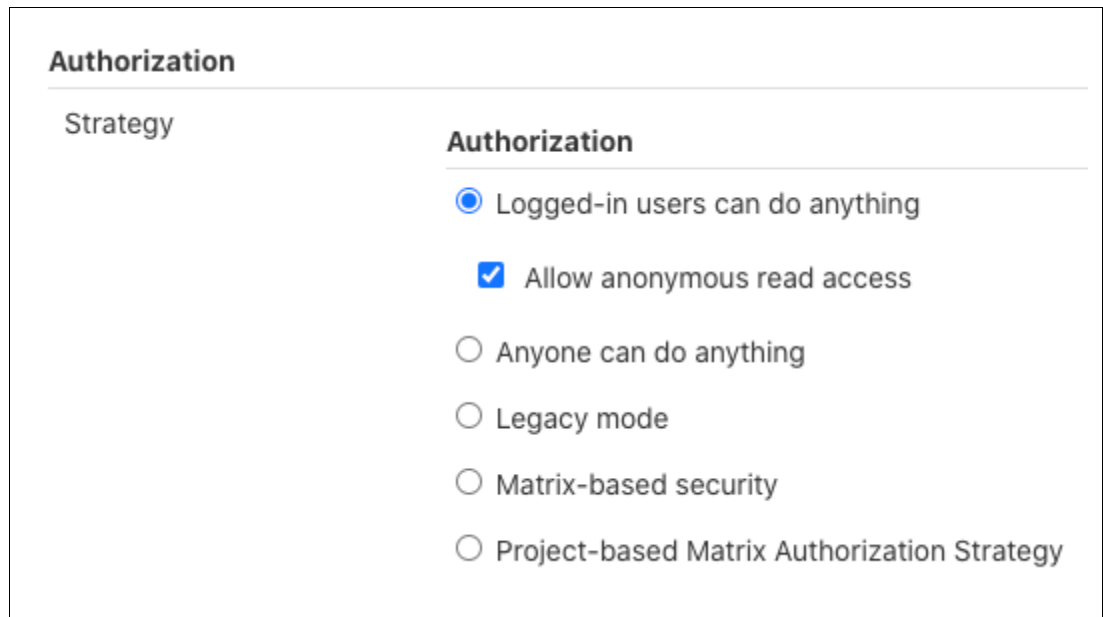
6. As shown in Figure 8-16, set the TCP port for inbound agents to 5000 and select **Fixed**.



The screenshot shows the Jenkins configuration page for 'Agents'. The 'TCP port for inbound agents' is set to 'Fixed' with the value '50000'. A red box highlights this section, and a red arrow points to it from the right. Other sections visible include 'Agent protocols...', 'CSRF Protection', 'Hidden security warnings', 'API Token', 'Agent -> Master Security', and 'SSH Server'.

Figure 8-16 Jenkins Set TCP Port

7. Enable **Allow anonymous read access** to allow requests to the Jenkins URL, as shown in Figure 8-17.



The screenshot shows the Jenkins 'Authorization' configuration page. The 'Strategy' is set to 'Logged-in users can do anything'. The 'Authorization' section shows 'Allow anonymous read access' checked. Other options include 'Anyone can do anything', 'Legacy mode', 'Matrix-based security', and 'Project-based Matrix Authorization Strategy'.

Figure 8-17 Jenkins Allow Anonymous Read Access

8.6.2 Building a Jenkins agent Docker image

In this section, we describe how to build the Docker image to run a Jenkins build agent. For more information about the files (including the Dockerfile) that are required to build and run the agent container, see Appendix A, “Additional material” on page 253.

Note: The Jenkins agent Dockerfile that is provided is customized with an installation of AdoptJDK, a Git client, Python, Ansible, Docker, and node.js.

Also, a system update and installation of required utility libraries is performed. All of these tools are preinstalled in the Docker image so that the agent container that is created from this image can be used for build or deploy and the tools need not be installed at each pipeline run from scratch.

Complete the following steps:

1. If not done so already, download the extra materials folder for DevOps and browse in the terminal to the folder in which the Jenkins agent files are stored. Run the following command to copy the file to your z/OS:

```
scp jenkins_agent.tar zcxadm3@wtsc74.pbm.ihost.com:/tmp
```

As shown in Example 8-14, you see the process of copying in your console.

Example 8-14 Output of scp jenkins_agent.tar ... command

```
base:Agent maik$ scp jenkins_agent.tar zcxadm3@wtsc74.pbm.ihost.com:/tmp
zcxadm3@wtsc74.pbm.ihost.com's password:
jenkins_agent.tar
100% 2996KB 129.9KB/s 00:23
```

2. After the copy process completes, your file is accessible in z/OS. Log in to your z/OS system by running the **ssh -l zcxadm3 wtsc74.pbm.ihost.com** command.
3. Browse to the /tmp folder where the jenkins_agent.tar file was copied by running the **cd /tmp** command.
4. To verify that the file was copied correctly, display files that contain jenkins_agent by running the **ls -l | grep jenkins_agent** command, which generates the output that is shown in Example 8-15:

Example 8-15 Output of ls -l | grep jenkins_agent command

```
ZCXADM3 @ SC74:/SC74/tmp>ls -l | grep jenkins_agent
-rw-r--r-- 1 MAIKEH SYS1 3067392 Jun 26 14:41 jenkins_agent.tar
```

5. Transfer the file securely by logging in to your zCX instance with **sftp** and running the following command:

```
sftp -P 8022 admin@129.40.23.72
```

You receive the output that is shown in Example 8-16 if a **sftp** connection was established successfully.

Example 8-16 Output of sftp -P 8022 admin@129.40.23.72 command

```
ZCXADM3 @ SC74:/SC74/tmp>sftp -P 8022 admin@129.40.23.72
Connected to 129.40.23.72.
```

6. To ensure the correct file format when copying files to another system, set file encoding to ASCII by running the **ascii unix** command.
7. Copy `jenkins_agent.tar` from z/OS to your zCX instance by running the following command:

```
put jenkins_agent.tar
```

As shown in Example 8-17, you see the progress in your console.

Example 8-17 Output of put jenkins_agent.tar command

```
sftp>put jenkins_agent.tar
Uploading jenkins_agent.tar to /home/admin/jenkins_agent.tar
jenkins_agent.tar
100% 2996KB 2.9MB/s 00:01
```

8. You now have the `jenkins_agent.tar` file stored in your zCX instance. Run the **exit** command to close the connection.
9. The next step is to log in to your zCX instance by running the following command:

```
ssh admin@129.40.23.72 -p 8022
```

You see the Welcome message that is shown in Example 8-18.

Example 8-18 zCX login welcome message

```
ZCXADM3 @ SC74:/u/zcxadm3>ssh admin@129.40.23.72 -p 8022
```

Welcome to the IBM z/OS Container Extensions (IBM zCX) shell that provides access to Docker commands.

For more information on how to use this shell to execute Docker commands refer to IBM

Last login: Fri Jun 26 14:23:27 2020 from 10.1.100.74

Sudo only permits specific User Management functions. See additional documentation for details.

10. Extract the content of `jenkins_agent.tar` by running the following command:

```
tar -xf jenkins_agent.tar
```

11. Go to the folder extracted from `jenkins_agent.tar` by running the **cd agent** command and check whether the `Dockerfile` is there by running the following command:

```
ls -l | grep Dockerfile
```

You see output that is similar to that which is shown in Example 8-19.

Example 8-19 Output of ls -l | grep Dockerfile command

```
admin@sc74cn05:~$ls -l | grep Dockerfile
-rw-r--r-- 1 admin admin 1298 Jun 24 17:50 Dockerfile
```

12. Because you now have the necessary files for building a Jenkins agent Docker image in your zCX instance, run the following build command:

```
docker build -f Dockerfile -t jenkins_agent .
```

The output you receive should contain the Docker image ID and the name tag (see Example 8-20).

Example 8-20 Output of docker build ... jenkins_agent . command

```
admin@sc74cn05:~$docker build -f Dockerfile -t jenkins_agent .  
Successfully built 86827336dc16  
Successfully tagged jenkins_agent:latest
```

13.If you set up a private registry in “Setting up a private registry” on page 174, tag the Jenkins agent Docker image to the registry by running the following command:

```
docker tag jenkins_agent localhost:5000/jenkins_agent
```

14.Push the Jenkins agent Docker image by running the following command:

```
docker push localhost:5000/jenkins_agent
```

Example 8-21 shows the output for the push command of the Jenkins agent Docker image to the registry.

Example 8-21 Output of docker push localhost:5000/jenkins_agent command

```
admin@sc74cn05:~$docker push localhost:5000/jenkins_agent  
The push refers to repository [localhost:5000/jenkins_agent]  
2950e536b743: Pushed  
826e5378699f: Pushed  
34f2ba2303ab: Pushed  
213f288f7dc8: Pushed  
32653909e039: Pushed  
bb2e8c728fbc: Pushed  
969be5474d21: Pushed  
dc710a402ff1: Pushed  
4a4819c7f7bc: Pushed  
b705bfa12379: Pushed  
8c3886cb258d: Pushed  
b07d92afb040: Pushed  
7d6a83014f9f: Pushed  
0c878cd793cb: Pushed  
cce965e350f9: Pushed  
202aae63ce0b: Pushed  
fc9f77e2034f: Pushed  
6a1551b59d48: Pushed  
d3c9262050e0: Pushed  
30525bf79200: Pushed  
ebaeae86b1e4: Pushed  
4c5823ae13d6: Pushed  
62d18496ede1: Pushed  
latest: digest:  
sha256:ff1c26e604eca450dafaba5f0965247352f11e431abde2a9f59ba8b3d2000623 size: 5151
```

If the image was pushed successfully into the private registry, you get the image name and tag by calling its API by running the following command:

```
curl 129.40.23.72:5000/v2/jenkins_agent /tags/list
```

The output to this command is shown in Example 8-22.

Example 8-22 Output of curl 129.40.23.72:5000/v2/jenkins_agent /tags/list command

```
admin@sc74cn05:~$curl 129.40.23.72:5000/v2/jenkins_agent /tags/list
{"name":"jenkins_agent","tags":["latest"]}
```

8.7 Running a Jenkins build agent Docker container

In this section, we describe three different options to run a Jenkins build agent Docker container:

- Option 1: Run a Jenkins build agent Docker container from a Dockerfile

If a Docker image is built in zCX, it can also be run from your local image cache in your instance. The container needs permission to access network interfaces, such as a Docker socket. For this reason, **--cap-add** must be included in the run command, as shown in Example 8-23. Also, it is essential to add the agent name and secret you created in 8.5, “Configuring Jenkins” on page 188.

Note: By default, zCX does not allow full read/write access to the Docker socket, the network interface to control the Docker engine. The read only mode (:ro) can be used as an extension to the Docker socket. The use of the read only mode as an extension to the Docker socket allows you to create containers without read/write access and is not refused by the zCX engine.

Example 8-23 Running a Docker image from local image cache

```
docker run -d --rm --name j-build-agent --cap-add ALL -v
/var/run/docker.sock:/var/run/docker.sock:ro jenkins_agent -url
http://129.40.23.72:3000
f1e53fa7fc611b4c9c8653468223ef06df3a0ac9060ca743a332df797689f968 j-build-agent
```

As shown in Example 8-24, the container ID is returned by running the **run** command.

Example 8-24 output of docker run ... j-build-agent command

```
admin@sc74cn05:~$docker run -d --rm --name j-build-agent --cap-add ALL -v
/var/run/docker.sock:/var/run/docker.sock:ro jenkins_agent -url
http://129.40.23.72:3000
f1e53fa7fc611b4c9c8653468223ef06df3a0ac9060ca743a332df797689f968 j-build-agent

45cc9bb53cdf4da633edd5e07dc561767f91010bb68505e9ac821c14c2e335f
```

- Option 2: Run a Jenkins build agent Docker container from a private registry
If you pushed your Jenkins agent image to the private registry, you can run the Jenkins agent image from the registry, as shown in Figure 8-25.

Example 8-25 Run the agent from the registry

```
docker run -d --rm --name j-build-agent --cap-add ALL -v  
/var/run/docker.sock:/var/run/docker.sock:ro localhost:5000/jenkins_agent -url  
http://129.40.23.72:3000  
f1e53fa7fc611b4c9c8653468223ef06df3a0ac9060ca743a332df797689f968 j-build-agent
```

You see the container ID as output in the console.

- Option 3: Run a Jenkins build agent Docker Container from dockerhub
If you want to start easily with running only a Jenkins agent container without building an image, you can pull and run the image from Docker Hub by running the following command:

```
docker run -d --rm --name j-build-agent --cap-add ALL -v  
/var/run/docker.sock:/var/run/docker.sock:ro maikohavemann/  
jenkins-agent-s390x:latest -url http://129.40.23.72:3000  
f1e53fa7fc611b4c9c8653468223ef06df3a0ac9060ca743a332df797689f968 j-build-agent
```

The image is downloaded and copied to your local system. If the **start** command was successful, the Container ID is displayed as an output in the console.

8.8 Checking Jenkins build agent Docker container status

By running the **docker ps** command, you can check whether the container runs and is configured correctly. You see output that is similar to what is shown in Example 8-26.

Example 8-26 Output of docker ps | grep j-build-agent command

```
admin@sc74cn05:~$docker ps | grep j-build-agent  
45cc9bb53cdf          maikohavemann/jenkins-agent-s390x:latest  
"/usr/local/bin/jenk..." 11 seconds ago    Up 10 seconds  
j-build-agent
```



Using Ansible to automate deployment and tests

This chapter provides instructions for writing Ansible playbooks and setting up an agent for deployment.

This chapter includes the following topics:

- ▶ 9.1, “Ansible overview” on page 206
- ▶ 9.2, “Setting up Ansible by using Jenkins” on page 206
- ▶ 9.3, “Setting up a Jenkins deployment agent” on page 208
- ▶ 9.4, “Running a Jenkins deploy agent Docker container” on page 210
- ▶ 9.5, “Checking Jenkins deploy agent Docker container status” on page 211

9.1 Ansible overview

Ansible is an automation engine that automates multiple tasks, such as cloud provisioning and multitier application deployment. Instead of writing many commands to deploy containers, such as in Jenkins, automation jobs in Ansible are described in YAML playbooks with which you can easily create a deployment. Ansible can be installed as a stand-alone tool or as a plug-in that is in your CI/CD pipeline tools as well.

Ansible connects your nodes and pushes out programs, called *Ansible modules*, to them. It starts these modules over **SSH** and removes them when the execution is finished.

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

9.2 Setting up Ansible by using Jenkins

In this section, we describe how to create Ansible playbooks for deployment to an environment and for integration tests of a simple `node.js` app. Ansible is set up in a Jenkins agent container. The playbooks can be found in Appendix A, “Additional material” on page 253.

For more information about how to create Ansible playbooks, see [this web page](#).

9.2.1 Creating an Ansible playbook for deployment to a development environment

In the first step, the Ansible framework is used to run the Docker image of the simple `nodejs` application in a development environment. This environment is simulated here by running the application on another port, but realistically can be a different network and Docker node.

On the agent that runs the deploy pipeline, you need the latest version of **pip** and **docker-py**.

After the container for the `node.js` Docker image is started, the image is pulled from a Docker registry. For more information, see Chapter 10, “Putting it all together and running the pipeline” on page 213.

The Ansible playbook to run the described procedure is shown in Example 9-1.

Example 9-1 Ansible Playbook for deployment to a dev environment

```
- name: build and run the docker container
  hosts: localhost
  tasks:
    - name: Install pip
      apt: name=python-pip state=present

    - name: install docker-py
      pip: name=docker-py

    - name: Creating the container
      docker_container:
        name: hello-node-dev
        image: localhost:5000/hello-node:latest
        state: present
```



```
- name: Running the container
  docker_container:
    name: hello-node-dev
    published_ports:
      - "3004:3000"
    image: localhost:5000/hello-node:latest
    state: started
```

9.2.2 Creating an Ansible playbook for integration tests

To test the deployment, the playbook that is shown in Example 9-2 runs a simple integration test.

Example 9-2 Ansible Playbook for simple integration test

```
- name: Simple Integration Test
  hosts: localhost
  tasks:
    - action: uri url=http://129.40.23.72:3004/ return_content=yes
      register: webpage
    - fail:
      msg: 'service is not happy'
      when: "'zCX' not in webpage.content"
```

The test sends an HTTP GET request to the application and expects “zCX” to be contained in the response. If the server returns anything else, for example, an error with status code 400 and empty body, the test fails. For more sophisticated testing, Ansible provides many tools and utilities.

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

9.2.3 Creating an Ansible playbook for deployment to a test environment

When integration tests passed, the third step is creating and starting the application in a test environment by using the playbook that is named `playbook-test`, as shown in Example 9-3. The steps of this stage are almost identical to those steps from the development playbook, except for the port of the running container.

Example 9-3 Ansible Playbook for deployment to a test environment

```
- name: build and run the docker container
  hosts: localhost
  tasks:
    - name: Install pip
      apt: name=python-pip state=present

    - name: install docker-py
      pip: name=docker-py

    - name: Creating the container
      docker_container:
        name: hello-node-test
        image: localhost:5000/hello-node:latest
```

```
state: present

- name: Running the container
  docker_container:
    name: hello-node-test
    published_ports:
      - "3003:3000"
    image: localhost:5000/hello-node:latest
    state: started
```

This test instance can then be used by testers for acceptance and other more tests before the application in the production environment is updated. For simplicity, we do not show this last step here because it is manual and highly dependent on the specific customer infrastructure and requirements.

9.2.4 Creating a deployment agent for Ansible in Jenkins

Jenkins allows you to provision environments and deploy applications with the use of shell scripts. Shell scripts can be cumbersome to maintain and reuse. When Ansible is used in the pipeline, it is possible to easily reuse playbooks for deployment and provisioning.

Jenkins can remain as an orchestrator and job scheduler instead of a shell script executor. Also, playbooks can be stored in a versioning system.

Jenkins allows use the use of Ansible as a plug-in and therefore can be integrated in a CI/CD flow. Because the Ansible plug-in was installed as described in 8.5.1, “Installing plug-ins manually” on page 194, a Jenkins agent for the Ansible deployment can be set up now.

9.3 Setting up a Jenkins deployment agent

The purpose of Jenkins is to automate jobs. Jobs are run by dedicated agents. To build the image of the `node.js` example application, an agent is set up. In this section, we describe how to set up a deployment agent.

9.3.1 Creating a Jenkins deployment node

The first step in setting up a Jenkins deployment agent is to create a Jenkins deployment node.

Complete the following steps:

1. Return to the Jenkins server user interface by accessing its URL (in our case, 129.40.23.72:3000). Then, click **Manage Jenkins** on the left side of the window.

2. As shown in Figure 9-1, select **Manage nodes and clouds**.

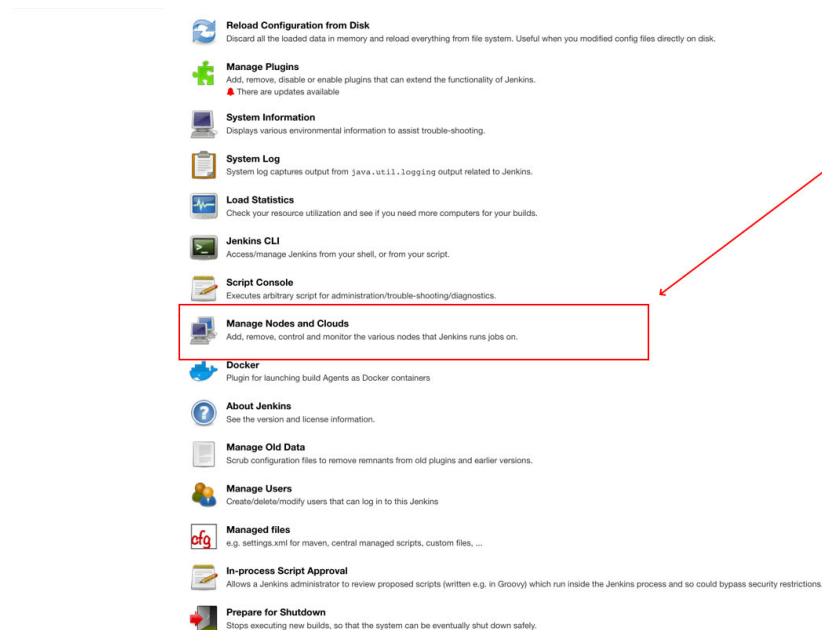


Figure 9-1 Jenkins Manage Nodes

3. Click **New Node**.

4. Create the agent as a permanent agent and configure it, as shown in Figure 9-2.

A screenshot of the Jenkins 'New Node' configuration form. The form is titled 'Name' and contains several fields and options. The 'Name' field is filled with 'j-deploy-agent'. The 'Description' field is empty. The '# of executors' is set to '1'. The 'Remote root directory' is set to '/home/jenkins/agent/'. The 'Labels' field is filled with 'test'. The 'Usage' dropdown is set to 'Use this node as much as possible'. The 'Launch method' dropdown is set to 'Launch agent by connecting it to the master'. There are checkboxes for 'Disable WorkDir', 'Custom WorkDir path', 'Internal data directory', 'remoting', 'Fail if workspace is missing', and 'Use WebSocket'. The 'Availability' dropdown is set to 'Keep this agent online as much as possible'. There is an 'Advanced...' button. At the bottom, there is a 'Node Properties' section with checkboxes for 'Environment variables', 'Tool Locations', and 'Disable deferred wipeout on this node'. A 'Save' button is at the bottom left.

Figure 9-2 Configure Deploy Node

5. You are provided with an overview of your agent. Copy the agent secret that is highlighted in Figure 9-3. You need this secret later to run the deploy agent container.



Figure 9-3 Jenkins Deploy Agent

9.4 Running a Jenkins deploy agent Docker container

In this section, we describe three different options to run a Jenkins deploy agent Docker container:

- Option 1: Run a Jenkins deploy agent Docker container from a Dockerfile

As described in 8.2, “Building a Jenkins server Docker image” on page 184, the Docker image that features all of the needed components for build and deployment jobs that are installed was built already. The container for the Jenkins deploy agent can be run by using the local build, as shown in Example 9-4.

Example 9-4 Running the Docker image from local image cache

```
docker run -d --rm --name j-build-agent --cap-add ALL -v
/var/run/docker.sock:/var/run/docker.sock:ro jenkins_agent -url
http://129.40.23.72:3000
34747350a14f981402816f91c731f7067860aa33923f1133e7064722c6c2e7e64 j-deploy-agent
```

As shown in Example 9-5, the container ID is returned by running the **run** command.

Example 9-5 output of docker run ... j-build-agent command

```
admin@sc74cn05:~$docker run -d --rm --name j-build-agent --cap-add ALL -v
/var/run/docker.sock:/var/run/docker.sock:ro jenkins_agent -url
http://129.40.23.72:3000
34747350a14f981402816f91c731f7067860aa33923f1133e7064722c6c2e7e64 j-deploy-agent
8e82ad1d98be97fa18c887d5f7884df223d05fe603d5d17e9125b63d2c5bfa05
```

- Option 2: Run a Jenkins deploy agent Docker container from a private registry

If you pushed your Jenkins agent image to the private registry, as described in “Building a Jenkins agent Docker image” on page 200, you can run the Jenkins agent image from the registry, as shown in Example 9-6.

Example 9-6 Run the agent from the registry

```
docker run -d --rm --name j-build-agent --cap-add ALL -v
/var/run/docker.sock:/var/run/docker.sock:ro localhost:5000/jenkins_agent -url
http://129.40.23.72:3000
34747350a14f981402816f91c731f7067860aa33923f1133e7064722c6c2e7e64 j-deploy-agent
```

Again, you see the Container ID as output in the console.

- Option 3: Run a Jenkins deploy agent Docker Container from dockerhub

If you want to easily start with running only a Jenkins agent container without building an image, you can pull and run the image from dockerhub, as shown in Example 9-7.

Example 9-7 Pull and run from dockerhub

```
docker run -d --rm --name j-deploy-agent --cap-add ALL -v
/var/run/docker.sock:/var/run/docker.sock:ro
maikahavemann/jenkins-agent-s390x:latest -url http://129.40.23.72:3000
34747350a14f981402816f91c731f7067860aa33923f1133e7064722c6c2e7e64 j-deploy-agent
```

The image is downloaded and copied to your local system. If the **start** command was successful, you see the container ID in the console.

9.5 Checking Jenkins deploy agent Docker container status

By running the **docker ps** command, you can check whether the container runs and is configured correctly. You should see output similar to what is shown in Example 9-8.

Example 9-8 Output of docker ps | grep j-deploy-agent command

```
admin@sc74cn05:~$ docker ps | grep j-deploy-agent
8e82ad1d98be          maikahavemann/jenkins-agent-s390x:latest
"/usr/local/bin/jenk..." 22 seconds ago    Up 22 seconds
j-deploy-agent
```



Putting it all together and running the pipeline

If you completed Chapter 7. “Using Gitea as a code repository”, Chapter 8. “Using Jenkins to automate builds” and Chapter 9. “Using Ansible to automate deployment and tests”, you are now ready to put them all together and run the pipeline.

This chapter includes the following topics:

- ▶ 10.1, “Pipeline architecture” on page 214
- ▶ 10.2, “Setting up a pipeline in Jenkins” on page 215
- ▶ 10.3, “Creating a webhook in Gitea” on page 220
- ▶ 10.4, “Running a pipeline in Jenkins” on page 223
- ▶ 10.5, “Monitoring a pipeline in Jenkins” on page 225
- ▶ 10.6, “Summary” on page 227

10.1 Pipeline architecture

After setting up the following components, the pipeline can be set up:

- ▶ Git as a code repository
- ▶ Jenkins server as pipeline orchestrator
- ▶ A Jenkins agent to build a Docker image
- ▶ A private registry to store that image
- ▶ A Jenkins agent including Ansible to deploy a node.js web application Docker container

The architecture overview that is shown in Figure 10-1 shows the DevOps flow that is described in this chapter.

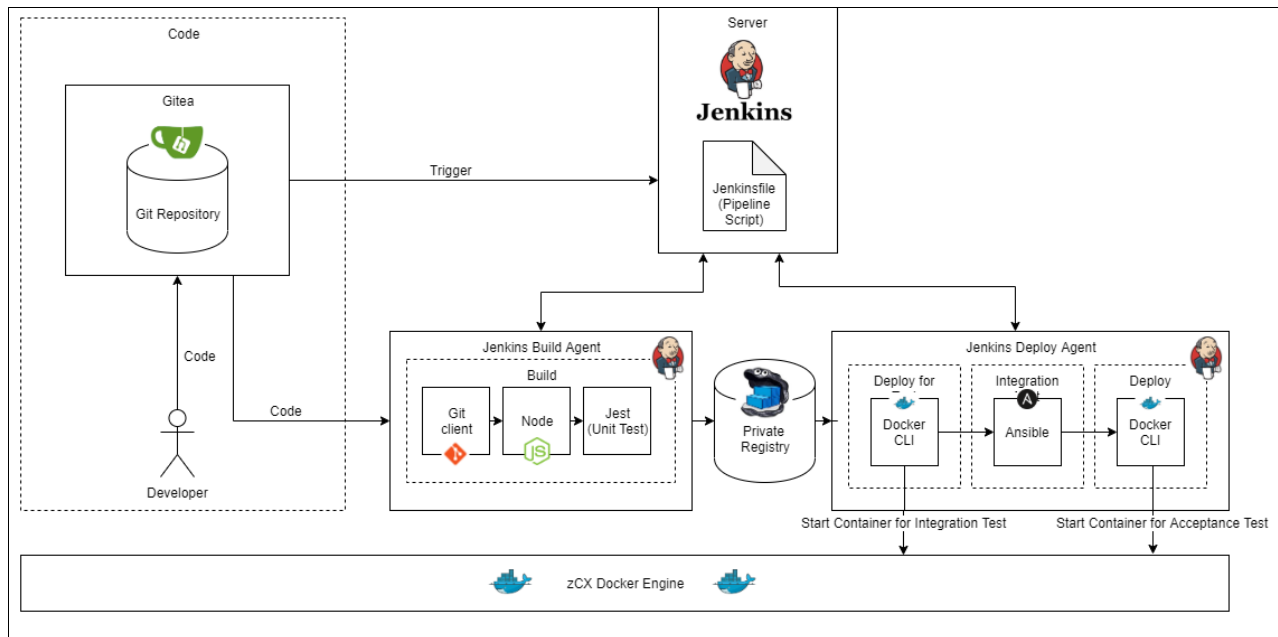


Figure 10-1 Pipeline Overview

10.2 Setting up a pipeline in Jenkins

Complete the following steps to set up a pipeline in Jenkins:

1. Browse to the Jenkins user interface URL (in our example, 129.40.23.72:3000). In the Jenkins home window, select **New Item**, as shown in Figure 10-2.

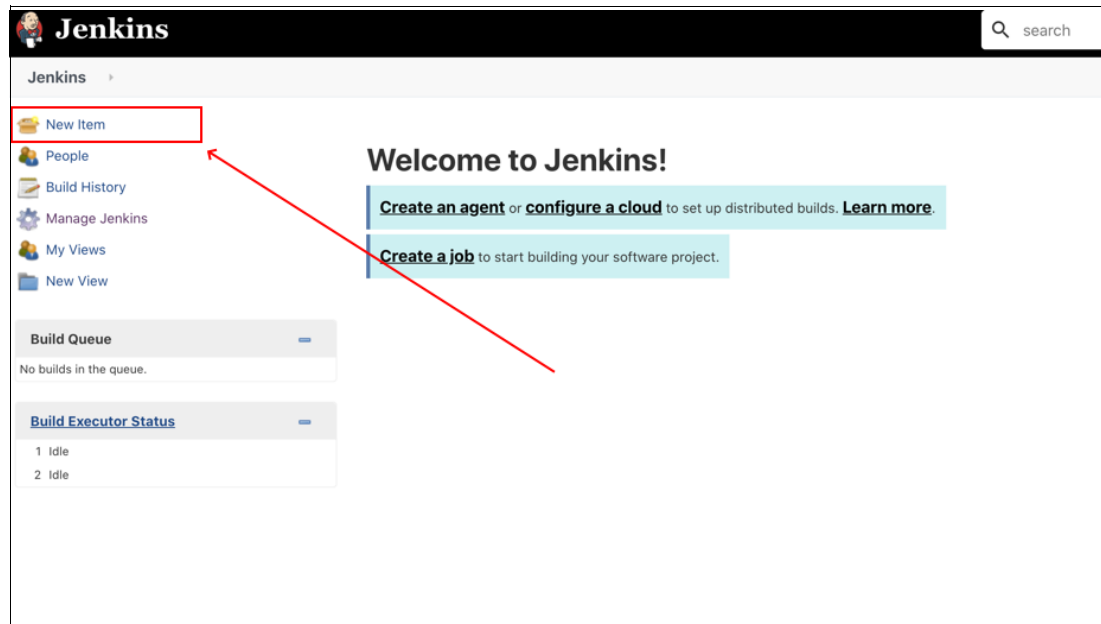


Figure 10-2 Selecting New Item option

2. Enter an item name and select **Pipeline** to create a pipeline, as shown in Figure 10-3.

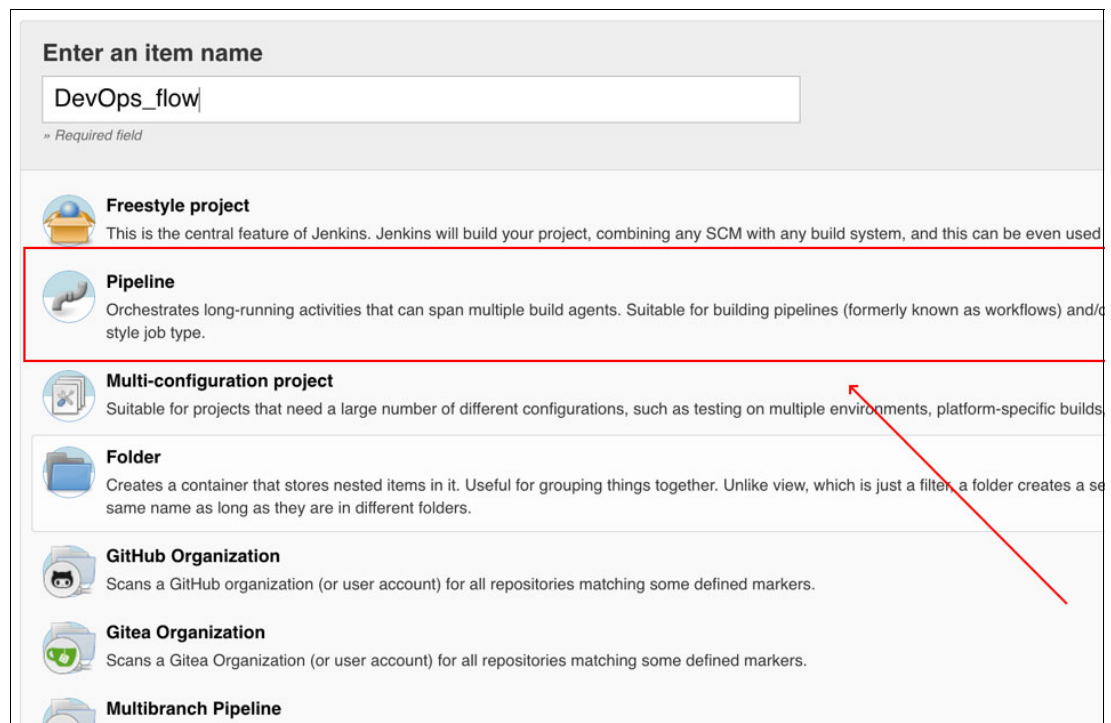
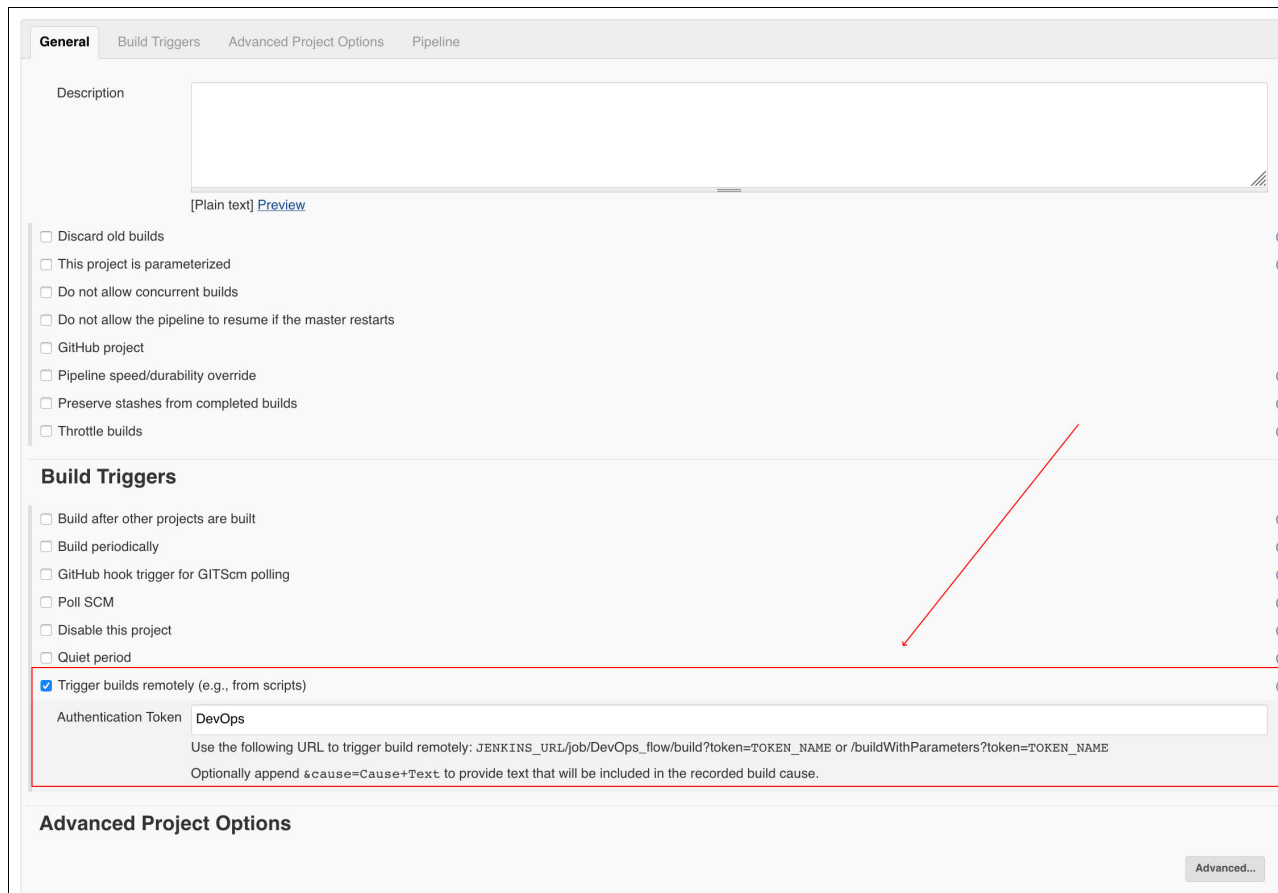


Figure 10-3 Jenkins selection: Pipeline

3. To ensure that the build and deployment pipeline is run whenever a change (push) to the code repository in Gitea is done, enable a build trigger and create an authentication token that is needed later. The configuration is shown in Figure 10-4.



The screenshot shows the Jenkins configuration page for a pipeline. The tabs at the top are 'General', 'Build Triggers', 'Advanced Project Options', and 'Pipeline'. The 'Build Triggers' tab is selected. The 'Description' field is empty. Below it, there are several checkboxes for build triggers, all of which are unchecked. The 'Trigger builds remotely (e.g., from scripts)' option is checked and highlighted with a red box. Below this, the 'Authentication Token' is set to 'DevOps'. A red arrow points to the 'Trigger builds remotely' option. The 'Advanced Project Options' section is visible at the bottom, with an 'Advanced...' button.

General Build Triggers Advanced Project Options Pipeline

Description

[Plain text] [Preview](#)

☐ Discard old builds

☐ This project is parameterized

☐ Do not allow concurrent builds

☐ Do not allow the pipeline to resume if the master restarts

☐ GitHub project

☐ Pipeline speed/durability override

☐ Preserve stashes from completed builds

☐ Throttle builds

Build Triggers

☐ Build after other projects are built

☐ Build periodically

☐ GitHub hook trigger for GITScm polling

☐ Poll SCM

☐ Disable this project

☐ Quiet period

☒ Trigger builds remotely (e.g., from scripts)

Authentication Token: DevOps

Use the following URL to trigger build remotely: JENKINS_URL/job/DevOps_flow/build?token=TOKEN_NAME or /buildWithParameters?token=TOKEN_NAME

Optionally append &cause=Cause+Text to provide text that will be included in the recorded build cause.

Advanced Project Options

Advanced...

Figure 10-4 Pipeline enable build trigger

4. As shown in Figure 10-5 on page 217, scroll down to the pipeline section where a pipeline script can be inserted. A script can be entered here or it can be retrieved from a source code management system (SCM).

The pipeline script in our case is called `Jenkinsfile` and is provided in Appendix A, “Additional material” on page 253 in the `hello-node` folder (included in the DevOps added material folder). If you used the provided `hello-node` files in 7.4.1, “Uploading code to Gitea” on page 179, the pipeline script is stored in the Gitea repository now.

In the pipeline definition, select **Pipeline script from SCM** and for the SCM type, select **Git**. Also, enter the repository URL from Gitea where the `Jenkinsfile` is stored. Then, click **Apply**. Our completed configuration is shown in Figure 10-5 on page 217.

Pipeline

Definition: Pipeline script from SCM

SCM: Git

Repositories:

- Repository URL: http://129.40.23.72:3008/maike/hello-node.git
- Credentials: - none -

Branches to build:

- Branch Specifier (blank for 'any'): */master

Repository browser: (Auto)

Additional Behaviours: Add

Script Path: Jenkinsfile

Lightweight checkout: ☒

Figure 10-5 Jenkins pipeline script definition

The Jenkinsfile runs a pipeline in stages, which includes one or more steps. A JSON-based structure organizes the mapping from pipeline or stage to an adequate agent and the order of the steps.

The Jenkins pipelines are sequential, meaning the order of single run steps is fixed. If one of those is unsuccessful (for example, returning a nonzero exit code), the entire pipeline stops. After each run, the pipeline can be in a state of SUCCESS (everything worked correctly and application was built and deployed) or FAILURE (something went wrong and the run was stopped). The pipeline script we used is shown in Example 10-1.

Example 10-1 Pipeline script to build and deploy a Docker container image

```
pipeline {
    // Disables automatic agent selection
    agent none

    stages {

        // Defines build stage containing all steps to
        // build and test a docker image for the nodejs application
        stage('Build') {

            // Select an agent that has the label 'build' attached
            // to run the steps of this stage
            agent{
                node {
                    label 'build'
                }
            }

            // Required steps for the build stage
            // (if one fails the whole stage fails)
            steps {
```

```

        // Clones the current data on master branch
        // configured in the pipeline configuration
        checkout scm

        // Install dependencies for the application
        sh 'npm install'

        // Run jest-based unit tests to check if all
        // interfaces are responding correctly
        sh 'npm test'

        // Build an docker image based on the files of the current directory
        // and pretag it with the address of you private registry
        sh 'docker build -t localhost:5000/hello-node:latest .'

        // Push the tagged image to the private registry
        sh 'docker push localhost:5000/hello-node'
    }
}

// Second stage for deployment to dev/test env and integration tests
stage('Deploy') {

    // Select an agent with the 'test' label attached
    // to run the steps of this stage
    agent{
        node {
            label 'test'
        }
    }

    // Execute several ansible playbooks to deploy the application
    steps{
        // Playbook to create and start an nodejs application container in dev
environment
        ansiblePlaybook(inventory: 'hosts.ini', playbook: 'playbook_dev.yaml')

        // Playbook to run integration tests that make api calls against the
application container
        ansiblePlaybook(inventory: 'hosts.ini', playbook: 'integration_test.yaml')

        // Playbook to create and start an nodejs application container in test
environment
        ansiblePlaybook(inventory: 'hosts.ini', playbook: 'playbook_test.yaml')
    }
}

// If boths stages and their corresponding steps are executed successfully,
// the pipeline finishes and displays "SUCCESS" in logs, else "FAILURE"
}
}

```

The script starts with a pipeline tag that denotes the starting point of the pipeline definition. Per default, Jenkins can run a pipeline on any available execution processor. For example, when the Jenkins server is initialized the first time, it has two build execution processors. In this scenario, the DevOps workflow is based on containers with scalability in mind. Hence, the run of the pipeline should happen on the agent container, which can easily be scaled up.

The agent selector of the pipeline is set to none, which overrides the default setting. The two stages define the groups of steps for build and deployment. Stages allow specific selection of an agent from the pool of available agents by using, for example, the build label attached. This way, agent containers with different preinstalled packages can be used for different steps and stages.

The build stage pulls the latest version of the application from the master branch of the Git repository. All files from the repository are stored temporarily in the agent container. Then, the required dependencies for the application are downloaded and installed.

For this step, the `sh` keyword can be used to run shell commands. The same keyword is used to run local tests with `npm`. A simple unit was created with the `jest` framework that allows starting a virtual web server locally to test APIs. In this scenario, the `hello-node` application has only one interface, which returns a simple text.

After successful unit tests, a new Docker image is created that is based on the currently downloaded and installed files. The tag of the image is composed of the address of our private registry (to be found under `localhost:5000`) and the name of the image (`hello-node`).

Also, the version tag, `latest` (which is set automatically if none is specified) is added. The tagged image is pushed to the private registry can be used in other agents and containers.

For deployment, the pipeline switches to a different agent that has the `test` label attached. For the steps of this stage, we use the preinstalled Ansible library and Ansible plug-in that is in Jenkins. It allows a structured definition of deployment and distribution tasks.

In Ansible, a set of tasks to run is called a *Playbook* and stored in YAML files. As described in 9.2, “Setting up Ansible by using Jenkins” on page 206 three playbooks are used for deploying the application in a development environment, the integration test, and deploying into a test environment. Because multiple environments are not used in this scenario, we simulate them with multiple containers and different ports.

10.3 Creating a webhook in Gitea

Complete the following steps to create a webhook in Gitea:

1. Return the Gitea URL (129.40.23.72:3008 in our example). Open the hello-node repository and click **Settings** in the upper right corner, as shown in Figure 10-6.

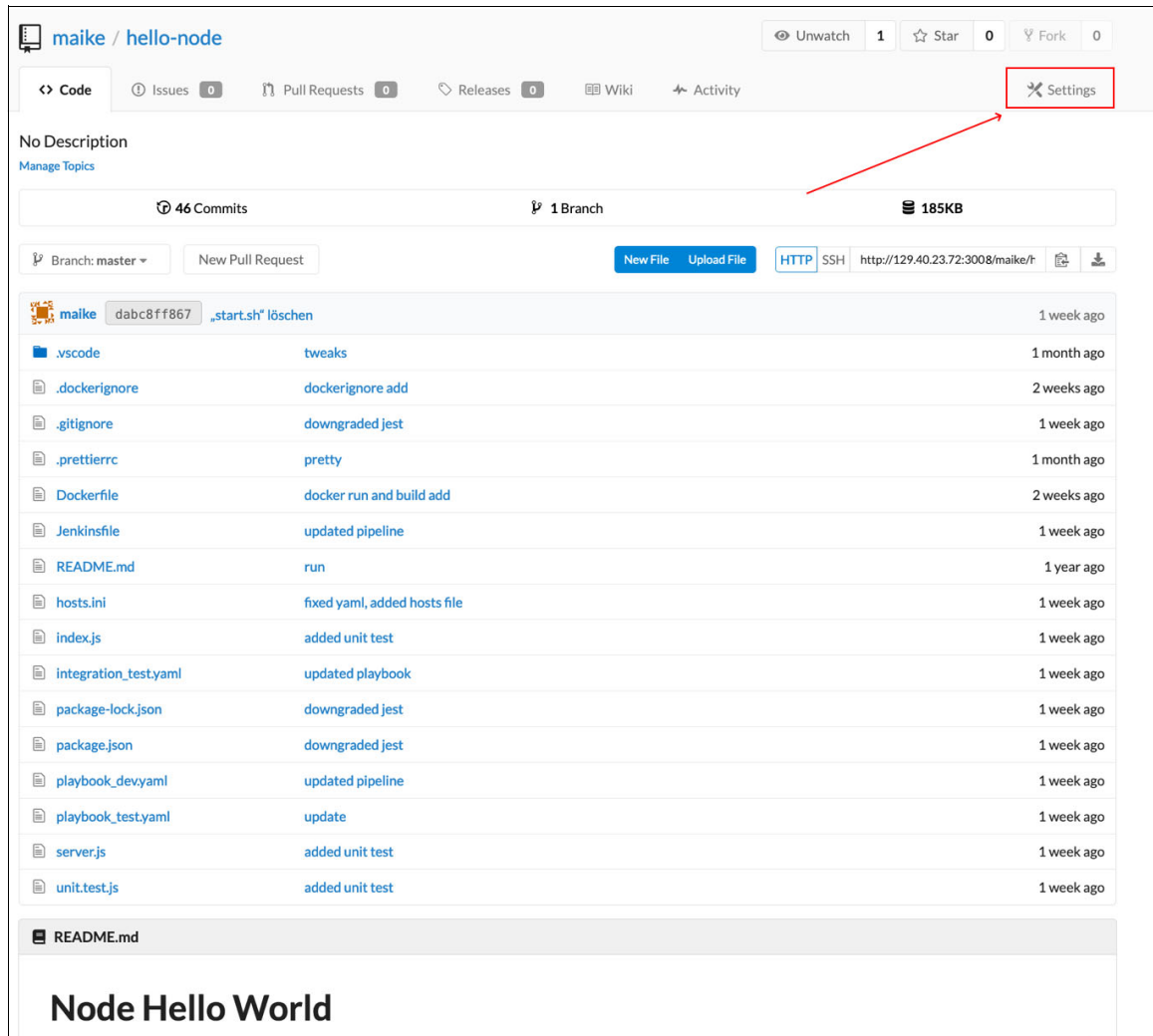


Figure 10-6 Gitea hello-node Repository

2. Select **Add Webhook** and then, choose **Gitea**, as shown in Figure 10-7.

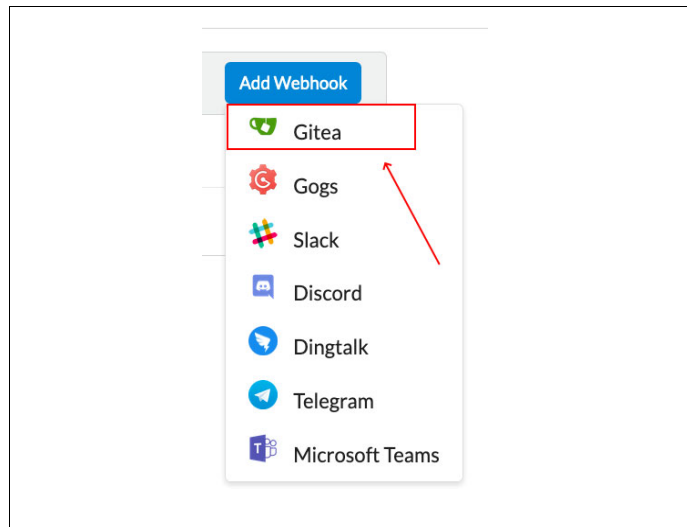


Figure 10-7 Add Gitea webhook

3. To create the webhook, the authentication token that was created as shown in Figure 10-4 on page 216 is needed. Enter the target URL and choose the content type as **POST**, as shown in Figure 10-8.

A screenshot of the 'Add Webhook' configuration form for Gitea. The form is titled 'Add Webhook' and includes a Gitea logo in the top right corner. It contains several fields and options: 'Target URL' with the value 'https://129.40.23.72:3000/job/DevOps_flow/build?token=DevOps', 'HTTP Method' set to 'POST', 'POST Content Type' set to 'application/json', a 'Secret' field, 'Trigger On' options with 'Push Events' selected, and a 'Branch filter' field with an asterisk. At the bottom, there is an 'Active' checkbox checked and an 'Add Webhook' button. A red arrow points to the 'Secret' field.

Figure 10-8 Configure Gitea Webhook

4. Change the HTTP Method in the drop-down menu, as shown in Figure 10-9, to **GET** and click **Add Webhook**.

Add Webhook

Gitea will send POST requests with a specified content type to the target URL. Read more in the [webhooks guide](#).

Target URL *

https://129.40.23.72:3000/job/DevOps_flow/build?token=DevOps

HTTP Method

GET

Secret

Trigger On:

- ☒ Push Events
- ☐ All Events
- ☐ Custom Events...

Branch filter

*

Branch whitelist for push, branch creation and branch deletion events, specified as glob pattern. If empty or *, events for all branches are reported. See github.com/gobwas/glob documentation for syntax. Examples: master, {master, release*}.

☒ Active

Information about triggered events will be sent to this webhook URL.

Add Webhook

Figure 10-9 Webhook HTTP Method

10.4 Running a pipeline in Jenkins

The pipeline is configured to start with every push and update to the hello-node repository in Gitea. Because no code update is available to push, the Test Delivery button in the webhook menu can be used (see Figure 10-10).

Update Webhook

Gitea will send POST requests with a specified content type to the target URL. Read more in the [webhooks guide](#).

Target URL *

`http://129.40.23.72:3000/job/DevOps_flow/build?token=DevOps`

HTTP Method

GET

Secret

Trigger On:

☒ Push Events

☐ All Events

☐ Custom Events...

Branch filter

*

Branch whitelist for push, branch creation and branch deletion events, specified as glob pattern. If empty or *, events for all branches are reported. See [github.com/gobwas/glob](#) documentation for syntax. Examples: master, {master, release*}.

☒ Active

Information about triggered events will be sent to this webhook URL.

Update Webhook **Remove Webhook**

Recent Deliveries

Test Delivery

2caba7c9-d4e4-48fa-8115-f04c1d1735bf 2020-07-03 14:47:26 UTC

Figure 10-10 Test a build trigger

Return to the Jenkins user interface (in our example, 129.40.23.72:3000) and click the pipeline project that was created in 10.1, “Pipeline architecture” on page 214.

Because a test delivery was started, the Jenkins pipeline now runs. The pipeline is divided in build and deploy. When the pipeline runs successfully, it looks similar to the example that is shown in Figure 10-11.

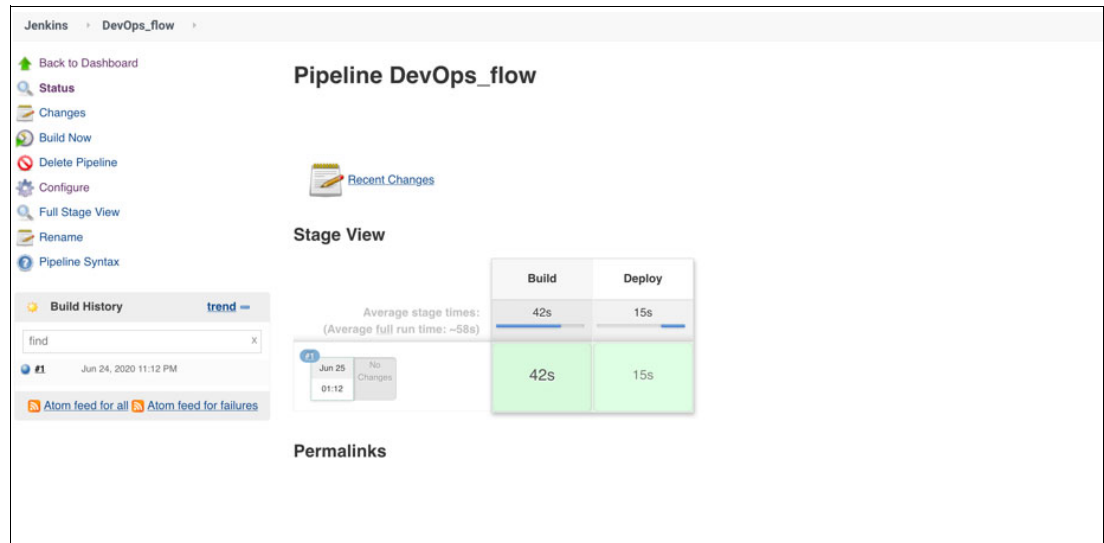


Figure 10-11 Jenkins pipeline progress

When visiting the URL to which Ansible deployed the application, the node.js application is displayed, as shown in Figure 10-12.

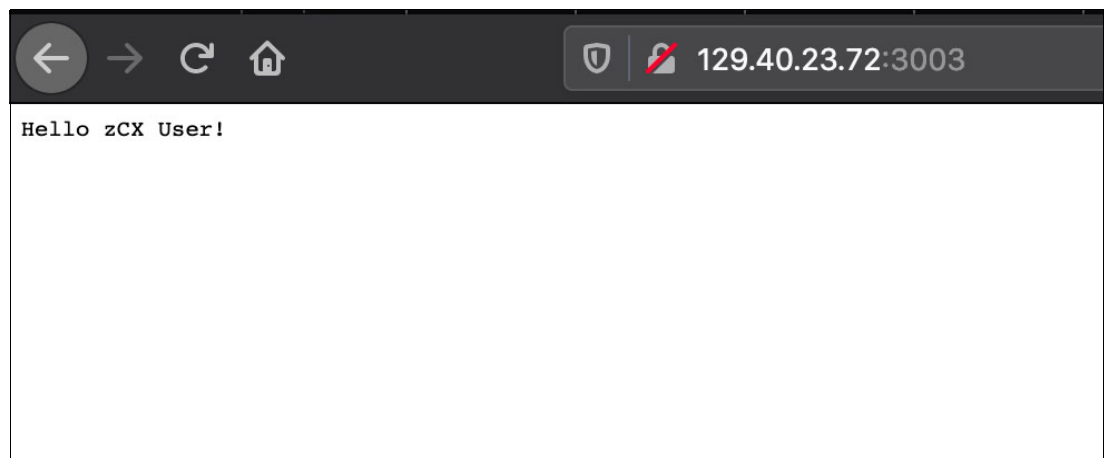


Figure 10-12 hello-node Application

10.5 Monitoring a pipeline in Jenkins

When a pipeline is created in Jenkins, it appears on the Overview page. Each pipeline is displayed as a row in a pipeline table.

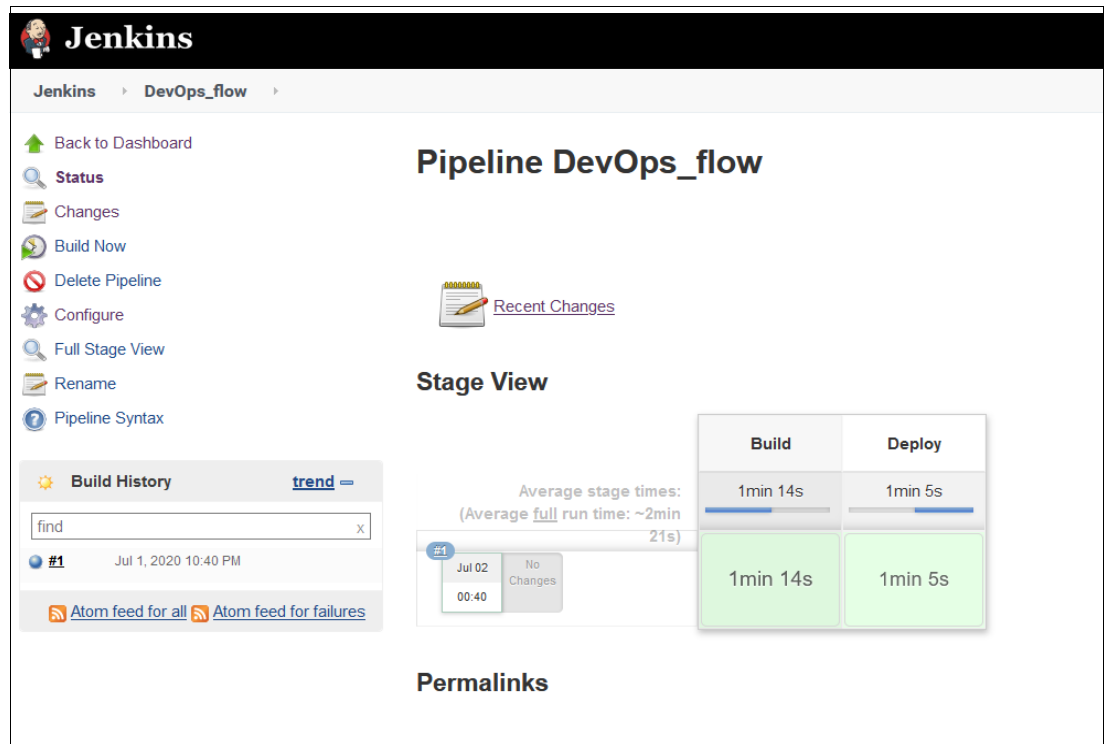
The first two columns visualize the health status of the pipeline; for example, the sun tells us that the last five pipeline runs were successfully completed. The first column refers to the state of the most recent run of the pipeline and is red if the pipeline fails (see Figure 10-13).



All	+					
S	W	Name ↓	Last Success	Last Failure	Last Duration	
		DevOps_flow	23 min - #1	N/A	2 min 21 sec	

Figure 10-13 Jenkins Pipeline Overview

After the pipeline in Jenkins is configured and ready, a build can be triggered by pushing a new commit to the master branch of the Git repository. Alternatively, you can run the pipeline immediately by clicking **Build Now** in the left side menu (see Figure 10-14).



The screenshot shows the Jenkins interface for the 'DevOps_flow' pipeline. On the left is a sidebar with navigation links: Back to Dashboard, Status, Changes, Build Now, Delete Pipeline, Configure, Full Stage View, Rename, and Pipeline Syntax. The main area is titled 'Pipeline DevOps_flow' and includes a 'Recent Changes' section. Below this is the 'Stage View' showing a progress bar for 'Average stage times' and 'Average full run time: ~2min 21s'. To the right of the stage view is a table comparing 'Build' and 'Deploy' stages.

Build	Deploy
1min 14s	1min 5s
1min 14s	1min 5s

Below the stage view is a 'Permalinks' section. In the bottom left corner, the 'Build History' section shows a list of builds, with the first build (#1) dated Jul 1, 2020 10:40 PM.

Figure 10-14 Jenkins Pipeline Status

In the center of the window, Stage View is an area that displays the status and progress of the stages that are defined in the pipeline.

In the lower left corner of Figure 10-14, you can see the build history. *Build* here means the run of the pipeline, not only a container build. You can click one of the items in the Build History list to see more information about a specific run.

In Figure 10-15, you can see the overview of a build history item. The date in the title refers to the start date of the pipeline run.

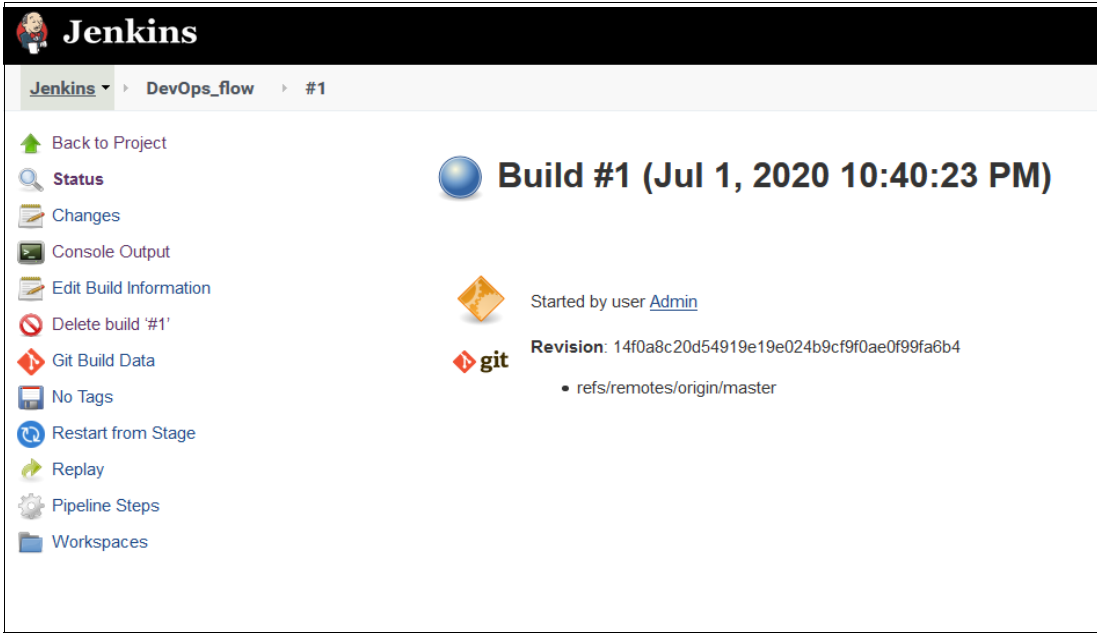


Figure 10-15 Build history of item

The build history shows who started the pipeline. Also, you can see the revision ID from the Git repository. On the left are various functions, such as status or console output. If the pipeline fails multiple times, it might help to review the console outputs to identify the root cause of the issue.

Figure 10-16 shows a typical snippet of a console output.

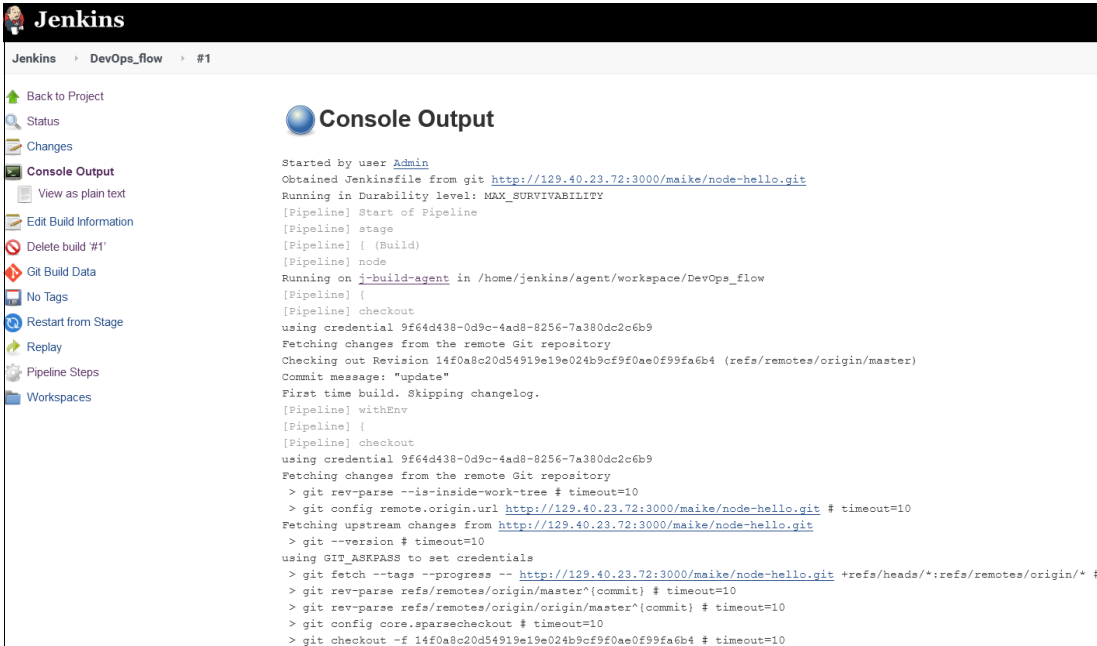


Figure 10-16 Pipeline Console Output information

In the output, meta-information and container output are displayed. The dark gray text is output from the pipeline runner and the container (container output features a “>” that is added as a prefix). Pipeline meta-information features a [Pipeline] that is added as a prefix.

Figure 10-17 shows example output from the Ansible playbooks. Because those playbooks are run and logged through the Ansible plug-in, the output is much more structured and shows the single tasks and their output.

```
PLAY [build and run the docker container] *****

TASK [Gathering Facts] *****
ok: [localhost]

TASK [Install pip] *****
ok: [localhost]

TASK [install docker-py] *****
ok: [localhost]

TASK [Creating the container] *****
ok: [localhost]

TASK [Running the container] *****
ok: [localhost]

PLAY RECAP *****
localhost : ok=5    changed=0    unreachable=0    failed=0    skipped=0    rescued=0    ignored=0

[Pipeline] ansiblePlaybook
[DevOps_flow] $ ansible-playbook integration_test.yaml -i hosts.ini

PLAY [Simple Integration Test] *****

TASK [Gathering Facts] *****
ok: [localhost]

TASK [uri] *****
ok: [localhost]

TASK [fail] *****
skipping: [localhost]

PLAY RECAP *****
localhost : ok=2    changed=0    unreachable=0    failed=0    skipped=1    rescued=0    ignored=0
```

Figure 10-17 Pipeline Ansible Output

10.6 Summary

In this chapter, we ran a simple DevOps flow by using Gitea, Jenkins, and Ansible in containers on zCX. zCX runs Docker images in a z/OS system.

This process demonstrates the potential for customers to modernize their IT by adopting a DevOps solution that helps them to become more agile and efficient while maintaining and adding features to z/OS applications. It also extends the value of z/OS assets to modern applications on IBM Z and to other platforms.

This capability is critical to running business applications in containers in a securable and cost-effective way. It also enables resiliency for critical workloads.



Part 3

Monitoring and managing zCX systems

In this part, we discuss monitoring and managing your zCX systems.

This part includes the following chapters:

- ▶ Chapter 11, “Monitoring” on page 231
- ▶ Chapter 12, “Container restart policies” on page 247



Monitoring

This chapter describes a monitoring use case that uses IBM Service Management Unite (SMU) Automation. The full Enterprise Edition also supports the IBM Workload Scheduler and IBM Omegamon in addition to IBM System Automation.

The benefits of running IBM SMU inside of zCX include the following examples:

- ▶ A modernized interface is used for several IBM management products.
- ▶ No external dependencies are needed.
- ▶ IBM SMU is available if the z/OS system is IPLed in an alternative location.

This chapter includes the following topics:

- ▶ 11.1, “IBM Service Management Unite” on page 232
- ▶ 11.2, “IBM Service Management Unite overview” on page 233
- ▶ 11.3, “Installation steps” on page 235
- ▶ 11.4, “Configuring the network and port” on page 236
- ▶ 11.5, “Starting the SMU image” on page 237
- ▶ 11.6, “Uninstalling Service Management Unite” on page 241
- ▶ 11.7, “Command wrapper overview” on page 243

11.1 IBM Service Management Unite

IBM Service Management Unite (SMU) Enterprise Edition is a customizable service management user interface that provides dashboards to monitor and operate IBM Z environments.

IBM Service Management Unite Enterprise Edition includes the following components:

- ▶ **IBM Service Management Unite Automation**
Provides the overall health status of the IBM NetView® domains, automation domains and nodes, and easy access to automation functions to start, stop, or recycle business resources.
- ▶ **IBM Service Management Unite Performance Management**
Helps you monitor and manage the performance of z/OS operating systems, network, storage, and other subsystems. You can quickly identify, isolate, and fix z/OS problems from a single point of control.
- ▶ **IBM Service Management Unite Workload Scheduler**
Provides comprehensive support for workload automation and scheduling. You can monitor and manage the most critical resources, such as jobs, job streams, and workstations in your scheduling environment.

These components work together to empower the operations staff to analyze and resolve problems more quickly.

IBM Service Management Unite Automation Enterprise Edition features the following capabilities:

- ▶ Provides a consolidated view of system health status, and thus reduces the time and effort in accelerating problem identification.
- ▶ Delivers simplified, efficient automation, workload scheduling, and network management capabilities, which streamlines operators' workflow.
- ▶ Provides an integrated operations console, which can be used to run commands and resolve problems. It also increases the degree of automation and avoids manual and time intensive tasks.
- ▶ Provides highly customized dashboard that helps you best suite your needs.
- ▶ Supports mobile access, which enables you to check your system anytime and anywhere.

11.1.1 IBM Service Management Unite Automation

IBM Service Management Unite Automation is a free of charge customizable dashboard interface that is available with the following platforms:

- ▶ IBM Z System Automation V4.1.0 or higher
- ▶ IBM Z Service Management Suite
- ▶ IBM Z Service Automation Suite
- ▶ IBM Z NetView V6.3 or higher
- ▶ IBM Z Monitoring Suite

It provides a single point of control for multiple System Automation Plexes (SAPlex) to operate in your environment. Operators can quickly and confidently analyze, isolate, and diagnose problems by providing all relevant data, including important logs in a single place.

IBM Service Management Unite Automation also enables operators to interact directly with the system by running commands and viewing results without going to a different console. Also, it enables customizing your own dashboards, which provides the information that is needed by the operations in your specific environment.

IBM Service Management Unite Automation can be installed on Linux on IBM Z and uses the end-to-end (E2E) automation adapter for secure communication with System Automation (SA) for z/OS.¹

11.2 IBM Service Management Unite overview

In this section, we provide an overview of a typical IBM SMU installation. In this example, two z/OS LPARs are running an end-to-end (E2E) adapter and both are connected to a single SMU instance, as shown in Figure 11-1.

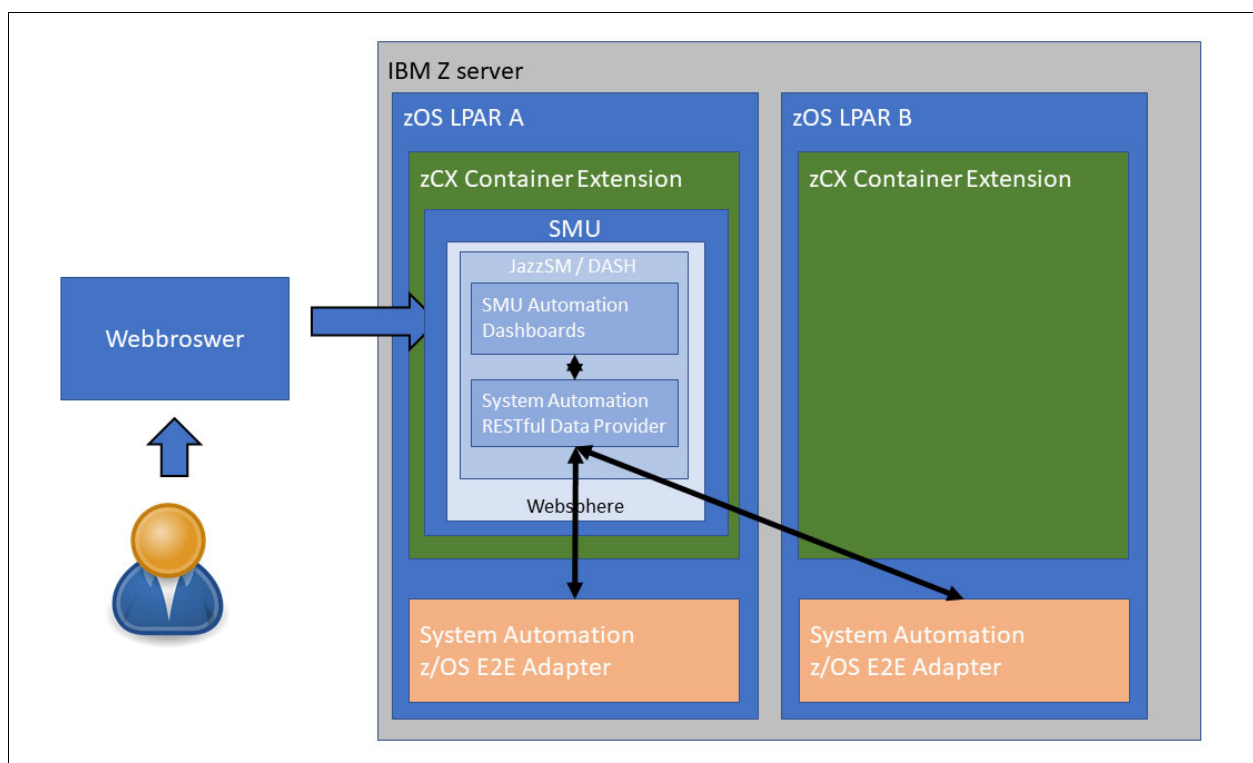


Figure 11-1 IBM SMU architecture

The benefit of the use of SMU is the ability to use a modernized interface to several IBM management products without any external dependencies. SMU remains available for the user, even when the SMU instance or the hosting LPAR is IPLed, because the instance can be moved to the next zCX instance where the E2E adapters connect to and offer the service seamlessly.

¹ https://www.ibm.com/support/knowledgecenter/SSWRCJ_4.1.0/com.ibm.safos.doc_4.1/UserGuide/Service_Management_Unite_Automation.html

The following single components are shown in Figure 11-1 on page 233:

► Service infrastructure

Service Management Unite Automation uses a service infrastructure that incorporates key products and services to run the dashboards and provide flexible integration and customization capabilities. The service infrastructure is provided with Service Management Unite Automation.

The infrastructure consists of the following components:

- IBM Dashboard Application Services Hub (DASH)/IBM Jazz® for Service Management (JazzSM)

IBM Dashboard Application Services Hub (DASH) provides visualization and dashboard services that are based on Jazz for Service Management (JazzSM). The DASH integration platform supports data processing and content rendering from multiple sources. The data is integrated and presented in interactive dashboards. DASH has a single console for administering IBM products and related applications.

- IBM WebSphere Application Server

IBM WebSphere Application Server provides the application server runtime environment for DASH and the Service Management Unite Automation dashboards.

- Service Management Unite Automation

Service Management Unite Automation provides the dashboards to monitor and operate resources that are automated by IBM Z System Automation, run z/OS and NetView commands, and access system logs. It also provides the Universal Automation Adapter to automate non-z/OS systems from IBM Z System Automation.

- Connectivity to backend systems
- Connect Service Management Unite Automation with z/OS systems

The following main components are used to interact with z/OS systems:

- IBM Z System Automation

IBM Z System Automation is a policy-based, self-healing, high availability solution. It maximizes the efficiency and the availability of critical systems and applications. It also reduces administrative and operational tasks.

- IBM Z System Automation end-to-end (E2E) adapter

The Z System Automation E2E adapter connects an SA z/OS domain to Service Management Unite Automation. It enables Service Management Unite Automation to read data, such as the status of automated resources, and run actions, such as sending requests.

It also provides the capability to issue NetView and z/OS commands and access system logs. In addition, the E2E adapter is used as the connection target by System Automation to provide cross-sysplex end-to-end automation.

- Connect Service Management Unite Automation with non-z/OS systems

Also, other methods are available to connect to non-z/OS systems with the setup of the Universal Automation Adapter (UUA). The UUA enables Service Management Unite Automation (SMU) to monitor, operate, and automate resources that are running on non-z/OS systems. It connects to the remote non-z/OS systems by using a Secure Shell (SSH).

11.3 Installation steps

Complete the following steps to install IBM Service Management Unite Automation:

1. Obtain the installation files:

- a. Go to the [download portal](#) to download IBM Service Management Unite Automation installation files (IBM ID log in required). To sign up for an ID, see [this web page](#).

The IBM Service Management Unite Enterprise Edition includes the following components that you install:

- IBM Service Management Unite Automation
- IBM Service Management Unite Performance Management
- IBM Service Management Unite Workload Scheduler

- b. Provide the access key to get the installation files.

The access key can be obtained by using one of the following methods:

- If APAR OA56692 is installed on IBM System Automation for z/OS, you can find the access key and more download information on your z/OS system in dataset SINGSAMP(INGESMU).
- Review the PDF that is supplied with the product materials on the CD titled *LCD8-2753-01 Accessing IBM Service Management Unite Automation CD-ROM*.

- c. Select the packages and click **Download now** to get the installation files.

- d. If you prefer to install IBM Service Management Unite Automation by using a prebuilt Docker image, select **IBM Service Management Unite Automation - Docker image for Linux on System z®**, depending on your system. The Docker image contains all of the software prerequisites that you need to install IBM Service Management Unite Automation.

2. Install IBM Service Management Unite Automation:

- a. Log on to the zCX instance and create the following smu directory:

```
mkdir smu
```

- b. Upload the image that was obtained as described in 11.2, “IBM Service Management Unite overview” on page 233 into the home directory of the admin user. Upload the image by using **SCP**.
- c. After the upload completes, the .tar file can be extracted into the newly created smu directory, as shown in Example 11-1.

Example 11-1 Extract the uploaded .tar file for SMU

```
admin@sc74cn11:~$ tar -xvf SMU_Automation_v1.1.7.1_Docker_Image_zLinux.tar
--directory /home/admin/smu
eezdocker.cfg
eezdocker.sh
readme_smu_auto.txt
smu_image_v1171.tar
util/
util/licenses/
util/licenses/LA_en
```

<output ommitted intentionally>

```
util/licenses/LA_in
util/migration.sh
```

```
util/basic_docker.sh
util/user_input.sh
util/messaging.sh
util/msg/
util/msg/en/
util/msg/en/sh_en.properties
```

After the extract process completes successfully, the .tar file is no longer needed. If space is limited, this file can be deleted.

11.4 Configuring the network and port

As a default configuration, the SMU Docker container uses a network bridge and maps the required ports to the Docker host system.

The Docker container's host name is set to the Docker host system's name. The Docker container opens and maps the ports (see Table 11-1) in listen mode for services that are offered by Service Management Unite.

Table 11-1 Default port information

Port number	Description
16311	Port to access the DASH that hosts the SMU dashboards.
16316	Port to access the WebSphere administrative console.
2002	Port that is used by automation adapters to connect to SMU and send update events for resources.
2005	Port that is opened by the Universal Automation Adapter to receive requests from the E2E agent.

If you want to restrict access to a port (for example, the WebSphere administrative console, port 16316), you must configure suitable firewall rules on the host system.

Note: A special configuration is required if you plan to use the Universal Automation Adapter (UAA) to manage resources that are running on the same host system where the SMU Docker container is running.

It is not possible to use the SMU Docker container host's host name as the node name in the UAA Policy because this host name is resolved to the Docker container's internal IP address by the SMU Docker container and not to the Docker host's IP address as required. To resolve this issue, you must use another host name other than the one that is set to the SMU Docker container, but resolving to the intended IP address.

In `eezdocker.cfg`, you can add `--add-host=<my_host_name>_host:<my_host_ip>` as an option to `DOCKER_NETWORK_CONFIG`. By using the Docker parameter `--add-host`, you can introduce the host name IP address mappings for a Docker container.

For example, if the host name where the SMU Docker container runs is `smu`, this introduces (inside the SMU Docker container) the host name `smu_host`, which resolves to the same IP address as `smu`. In the UAA's policy, you specify the resource's node to this new host name, `smu_host`.

11.5 Starting the SMU image

Before starting the image, the image must be loaded, as shown in Example 11-2. The license agreement must be accepted, by selecting option 1 as indicated in [blue](#).

Example 11-2 Load the SMU image

```
admin@sc74cn11:~$ /home/admin/smu/eezdocker.sh load
+++++
+          IBM Service Management Unite Docker Command Line Utility          +
+++++

Executing command load ...

Docker image smu_auto:1171 ----- Not loaded.
You must accept the license agreement before the IBM Service Management Unite
Docker image can be loaded.
To read the license agreement select 3 and toggle between the license files with
:n (next) and :p (previous).
Exit the reader when you have finished with q.
If you want to accept the license, continue with 1.
If you want to decline the license, cancel with 2.
[Waiting for response] Accept license [ 1-accept or 2-decline or 3-view; decline
is default ]? 1
License accepted. Continue loading the IBM Service Management Unite Docker image.
Loading image from local file /home/admin/smu/smu_image_v1171.tar.
550cfc7b1d35: Loading layer [=====>]
66.07MB/66.07MB
0cf58cc46810: Loading layer [=====>]
991.7kB/991.7kB
bbcd7c7060c0: Loading layer [=====>]
15.87kB/15.87kB
116be52951b9: Loading layer [=====>]
3.072kB/3.072kB
f7ccbe30c7ea: Loading layer [=====>]
490.2MB/490.2MB
d88865165dca: Loading layer [=====>]
249.9MB/249.9MB
9bca0be1df95: Loading layer [=====>]
2.496GB/2.496GB
65bca1e023a3: Loading layer [=====>]
39.65MB/39.65MB
59b1e70038fe: Loading layer [=====>]
10.24kB/10.24kB
d7b372ef49e6: Loading layer [=====>]
5.12kB/5.12kB
73626721a8a1: Loading layer [=====>]
2.56kB/2.56kB
4c7a4f0ed798: Loading layer [=====>]
201.4MB/201.4MB
979088069557: Loading layer [=====>]
444.3MB/444.3MB
d9470ba6ffe4: Loading layer [=====>]
124.7MB/124.7MB
Loaded image: smu_auto:1171
```

Loading Docker image smu_auto:1171 ----- Successful.

To start the image, run the command that is shown in Example 11-3. As shown, the **eezdocker.sh** script first checks whether the container is running and then, whether the image is available to be started.

Example 11-3 Start the SMU image for the first time

```
admin@sc74cn11:~$ /home/admin/smu/eezdocker.sh start
+++++
+          IBM Service Management Unite Docker Command Line Utility          +
+++++
```

Executing command start ...

```
Status of Docker container smu_auto_1171 ----- Not started.
Docker container smu_auto_1171 ----- Not available.
Docker image smu_auto:1171 ----- Loaded.
Creating Docker container smu_auto_1171 ----- Successful.
Starting Docker container smu_auto_1171 ----- Successful.
Status of Docker container smu_auto_1171 ----- Started.
```

If the **start** command is not run the first time, the output differs because the container is already available (see Example 11-4).

Example 11-4 Start the SMU image not the first time

```
admin@sc74cn11:~$ /home/admin/smu/eezdocker.sh start
+++++
+          IBM Service Management Unite Docker Command Line Utility          +
+++++
```

Executing command start ...

```
Status of Docker container smu_auto_1171 ----- Not started.
Docker container smu_auto_1171 ----- Available.
Starting Docker container smu_auto_1171 ----- Successful.
Status of Docker container smu_auto_1171 ----- Started.
```

Example 11-5 shows what occurs when the start parameter is used when SMU is already running.

Example 11-5 Start the SMU image when already started

```
admin@sc74cn11:~$ /home/admin/smu/eezdocker.sh start
+++++
+          IBM Service Management Unite Docker Command Line Utility          +
+++++
```

Executing command start ...

```
Status of Docker container smu_auto_1171 ----- Started.
```

After SMU is started, it might take a few seconds until the built-in IBM WebSphere Application Server is started. After it is started, you can access the SMU Dashboard with the following link:

`https://<zCX_DNS>:16311/ibm/console`

Accessing the website by using a browser shows a log in page similar to the example that is shown in Figure 11-2.

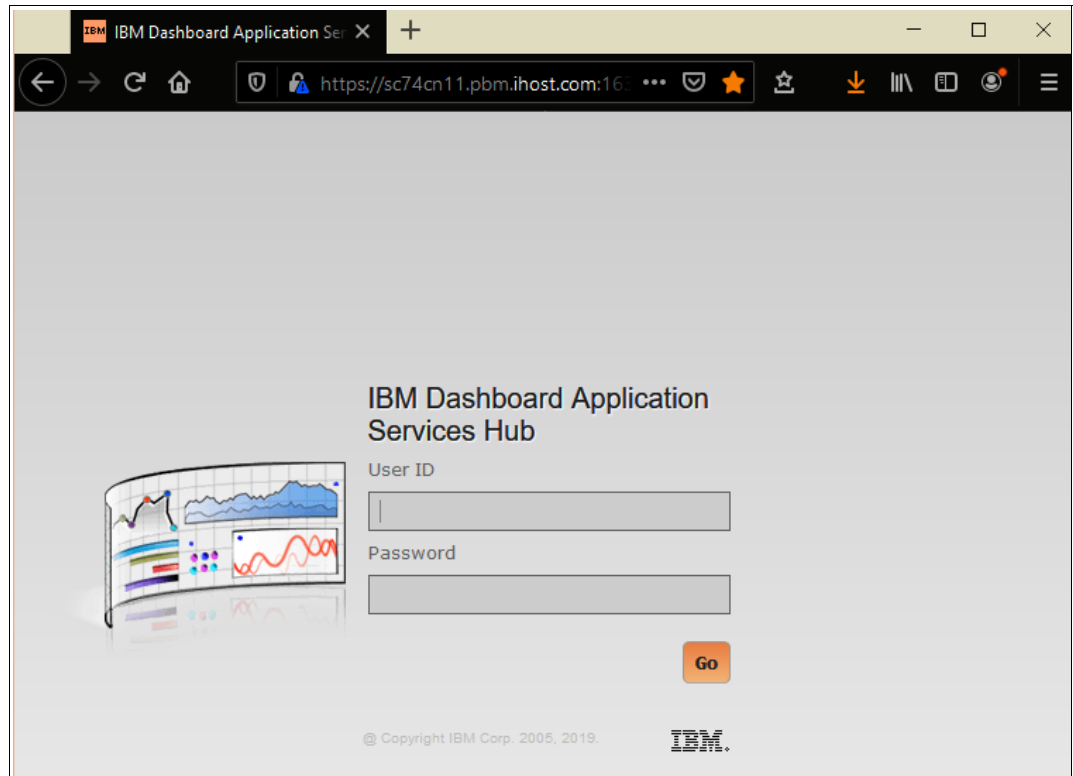


Figure 11-2 IBM SMU Dashboard log in page

The following default credentials are used to log in (these should be changed immediately after installation):

- ▶ User ID: eezadmin
- ▶ Password: eezadmin

Use one of the following methods to access the WebSphere Administrative Console:

- Use the following link:
`https://<zCX_DNS>:16316/ibm/console`
- When logged in to the Dashboard, click the **Console Settings** icon in the upper right corner of the window and select **WebSphere Administrator Console**, as shown in Figure 11-3.

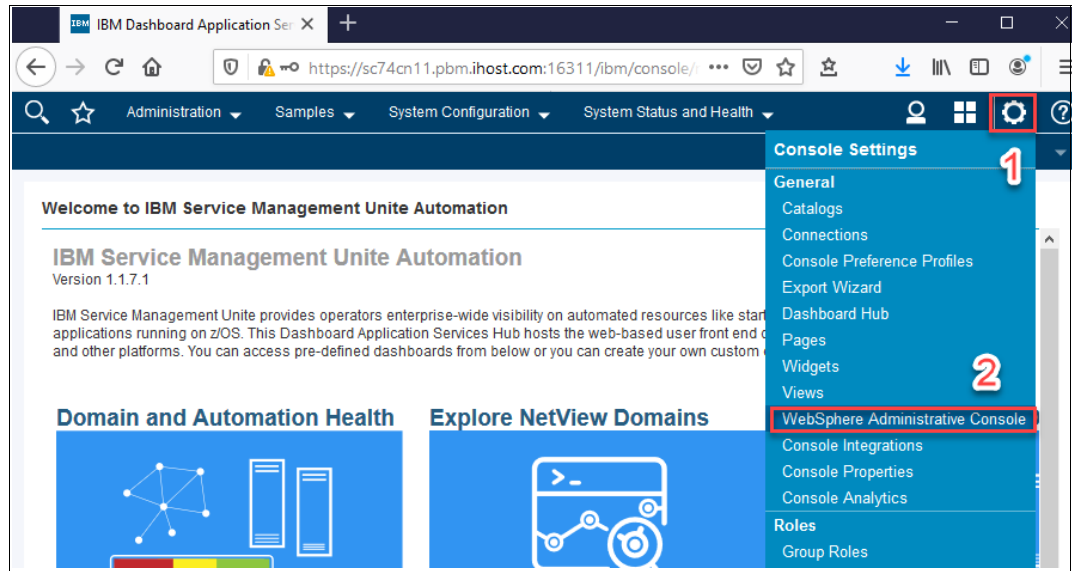


Figure 11-3 Accessing WebSphere Administrative Console

From the main menu of the IBM Service Management Unite Automation, you find information from all connected System Automation Plexes (SAPlex). The configuration information and all prerequisites are available at [IBM Knowledge Center](#).

When the SA configuration is completed and you select **Domain and Automation Health**, you see an overview page in the connected Domains and Nodes window, as shown in Figure 11-4.

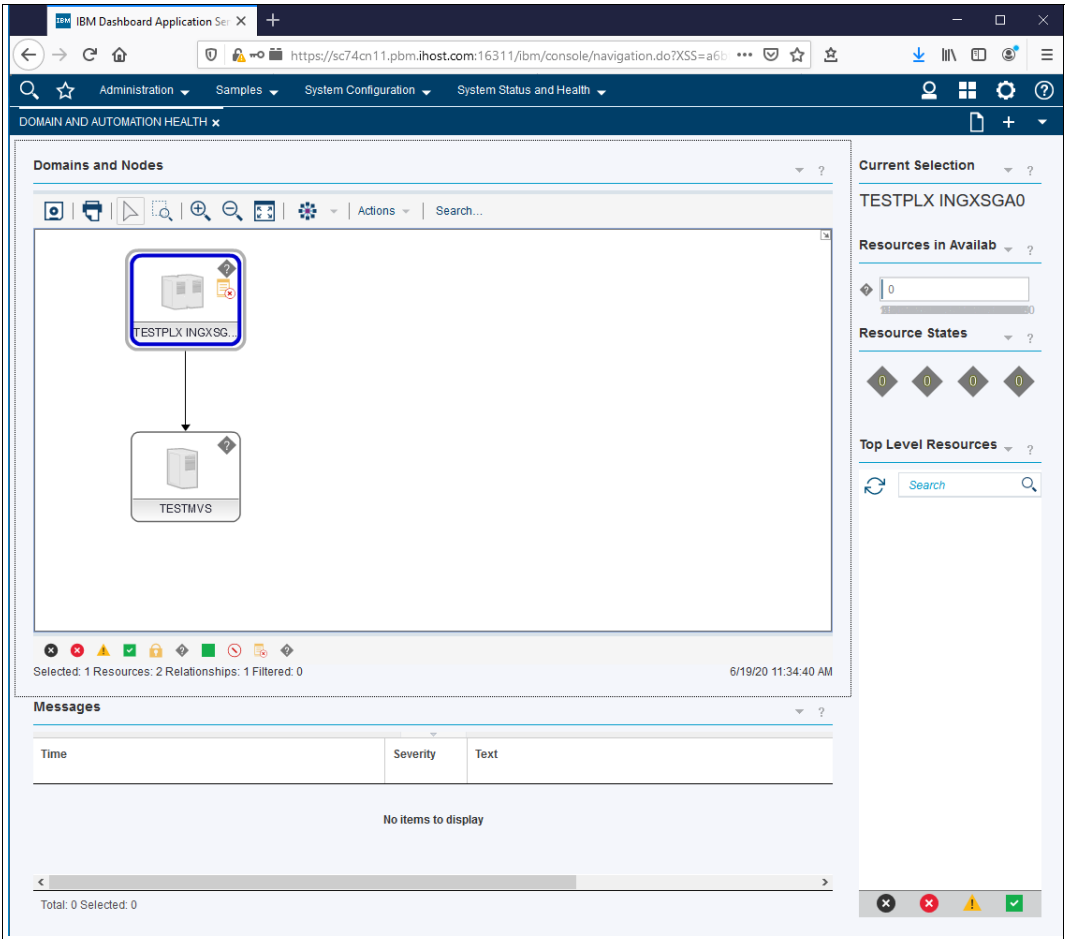


Figure 11-4 SMU domain and automation health

11.6 Uninstalling Service Management Unite

To uninstall Service Management Unite, remove the SMU Docker image and all SMU Docker containers from the host system.

Complete the following steps:

1. Any SMU Docker container must be stopped before the uninstallation. Run the command that is shown in Example 11-6 to stop the SMU Docker container.

Example 11-6 Stop the SMU image

```
admin@sc74cn11:~$ /home/admin/smu/eezdocker.sh stop
+++++
+          IBM Service Management Unite Docker Command Line Utility          +
+++++

Executing command stop ...
```

```
Status of Docker container smu_auto_1171 ----- Started.
Stopping Docker container smu_auto_1171 ----- Successful.
Status of Docker container smu_auto_1171 ----- Not started.
```

2. Run the command that is shown in Example 11-7 to remove the SMU Docker container and Docker image. Confirm the deletion, as shown in **red** in the example to remove all customer data.

Example 11-7 Uninstall SMU

```
admin@sc74cn11:~$ /home/admin/smu/eezdocker.sh uninstall
+++++
+          IBM Service Management Unite Docker Command Line Utility          +
+++++

Executing command uninstall ...

Uninstalling SMU from the Docker environment will delete the existing Docker
image smu_auto:1171 and container smu_auto_1171.
[Warning] ALL CUSTOM CONFIGURATION WILL BE LOST!
[Waiting for response] Perform deletion [ 1=yes or 2=no; no is default ]? 1
Confirmed. Continue with deletion.
Docker container smu_auto_1171 ----- Available.
Status of Docker container smu_auto_1171 ----- Not started.
Deleting Docker container smu_auto_1171 ----- Successful.
Docker image smu_auto:1171 ----- Loaded.
Untagged: smu_auto:1171
Deleted:
sha256:740fa7b486975dcc7973423841d5fbc8a71dfadf80bfc15cb7a550123f38bb32
Deleted:
sha256:c0da71e660f34f0e9eaa15bdac7b33f8a868114477b515c7fab621b11eb2533b
Untagged: smu_auto:1171_running
Deleted:
sha256:c03943cef5927a24ba3a094ee360ba9981033b0301d93834a1aa1b7b1882de87
Deleted:
sha256:066b48e8effdcec5ab8b3aceb1d63da93bbf4144db00e2389557052356261cfa
Deleted:
sha256:738b80f0e0afe37ccc26630bae3917d3e156f68ae6dac6f88f2c781b89aa3105
Deleted:
sha256:b221b75c47c418c6f191f4be5c7475fd7a3681cfecce7918edb1bbe5f3738cc7
Deleted:
sha256:4a06aba3b0ab51d5fc5a1b44d5854b7464b94b493d92a00aee32776b368bd95d
Deleted:
sha256:782eef30a4e8f3bd5861b1e2f72ce73a127db4cbfedc316666763029b4c802ca
Deleted:
sha256:0151fa8e79f90324d6031d4d65b9b9799139c56e7cbfbd72ab132306ea35b88a
Deleted:
sha256:6716c502a1e1559635a262a74bdcba976dbc6c1617785672a5d0621da306425
Deleted:
sha256:89ae5addd42e7ed249c086f5c091198d40b6d595ae7c36075dfbe3e462bf019d
Deleted:
sha256:78b2859038ec6692ba741262a34bb2160593d4b47b3d4f8470a791049602e958
Deleted:
sha256:59f3e4529c2e2a561cab09b256a984da41418120a7d672f8369a7b25e0284eeb
Deleted:
sha256:531c74dcedc5d4466c63dede17cf7913c78198961bae2def3efcdade181958ded
```

```
Deleted:
sha256:20b31cdded976b3672b30e9f86f2f6654916c44abafbdb6976a302c03ec34f54
Deleted:
sha256:550cfc7b1d35a3a70b83418a9a869e2ac1e41d7939c6ccc6d442e4a43620c223
Deleting Docker image smu_auto:1171 ----- Successful.
Uninstalling SMU from the Docker host ----- Successful.
```

3. The SMU instance is now removed. For the final cleanup, remove the entire `smu` directory that was created in Step 2., “Install IBM Service Management Unite Automation:” on page 235.

11.7 Command wrapper overview

A command wrapper available to manage an SMU instance.

When you download the SMU Docker archive, extract the package, and initially run command **`eezdocker.sh load`**. The SMU Docker image is loaded into your local Docker environment.

The image is like a blueprint for you and it includes SMU and all its prerequisites but is missing your custom configuration. To use SMU in your environment, you must create a Docker container from the SMU Docker image. A container is a concrete, runnable instance of an image. Theoretically, you can create more than one container instance from the same image in parallel and configure each container individually.

When you first run the **`eezdocker.sh start`** command, the SMU Docker Command Line Utility automatically creates a SMU Docker container from the SMU Docker image for you. The created Docker container can be started and stopped as often as you like and also survives from a restart of the Docker environment or a restart of the host system.

Unless you specifically delete the SMU Docker container (**`eezdocker.sh reset`** or **`eezdocker.sh uninstall`**), the SMU Docker Command Line Utility operates on the same SMU Docker container instance. For example, if you stop the SMU Docker container (**`eezdocker.sh stop`**) and start it again (**`eezdocker.sh start`**), it is the same SMU Docker container instance.

Every SMU and WebSphere Application Server configuration change that you perform on a running SMU Docker container is stored within this container instance, but not in the SMU Docker image. For example, if you create a custom dashboard in DASH, the change is stored within the SMU Docker container and is there until the container is deleted. The dashboard also is still there if the SMU Docker container is restarted or even if the host system is restarted.

Therefore, the easiest way to reset SMU to factory defaults is to delete the SMU Docker container and create a one from the SMU Docker image (**`eezdocker.sh reset`**).

In SMU V1.1.4, Docker volumes are used to store the SMU and WebSphere Application Server configuration outside of the SMU Docker container in a specific directory on the host system. From SMU V1.1.5, the SMU Docker container no longer uses any Docker volumes. All configuration is stored within the SMU Docker container and is not yet lost unless you delete the container.

In addition, a migration command is provided with which you can migrate your custom configuration from an SMU Docker container of an old version to a new version.

11.7.1 SMU Docker command line utility commands

The following useful commands can be used with the command line utility:

Important: Because the path that is used to upload the files (in our example, /home/admin/smu) is not part of the \$PATH configuration of the zCX Instance, the **eezdocker.sh** script must be called with an absolute or relative path, such as **./eezdocker.sh <parm>** or by adding the /home/admin/smu path into the \$PATH list.

- eezdocker.sh load** Loads the IBM provided SMU Docker image into the local Docker environment. This command is run only once. If the image is successfully loaded, the .tar file can be deleted.
- eezdocker.sh start** Starts the SMU Docker container if it is stopped. If no SMU Docker container exists, it creates a SMU Docker container from the loaded SMU Docker image.
- eezdocker.sh stop** Stops the running SMU Docker container.
- eezdocker.sh restart** Restarts the running SMU Docker container. It stops the SMU Docker container and restarts it. For example, the command can be run if restart the WebSphere Application Server after a configuration change.
- eezdocker.sh shell** Opens a Bash shell to the running SMU Docker container. It allows access the internal of the container; for example, if the configuration files must be edited manually. To exit the shell, run **exit** command in the shell. It exits only the shell connection into the SMU Docker container, and the container and SMU continue to run.
- eezdocker.sh cfgsmu** Starts the SMU Automation **cfgsmu** tool and sets up the necessary X-Forwarding so that the tool's GUI can be displayed with the host's Window Manager.
- If **cfgsmu** cannot be run out of the Docker container, it might be necessary to allow access to the X11 session on the host system. Run the **xhost+local:a11** command before running **eezdocker.sh cfgsmu** to ensure that the Docker process can access the user's X session.
- eezdocker.sh collect_logs**
- Collects and bundles all relevant log files from the running SMU Docker container and copies them to the host systems /tmp folder.
- For example, run this command if a problem occurs and the IBM Support team requests the log information.
- eezdocker.sh reconfigure**
- Some Docker configurations can be specified only during the creation of the SMU Docker container; for example, the network configuration. If you must change a configuration option, run this command so that the configuration changes take effect.
- Internally, the current SMU Docker container is committed to a snapshot of the SMU Docker image, from which a container is created. The new container has all of the same configurations and customizations of the old container, but runs with the new configuration.

When you run the **eezdocker.sh reconfigure** command, a snapshot is created. Keep the snapshot image because an image cannot be removed if containers are derived from it. The image does not use too much disk space because only the changes (compared to the official SMU Docker image) are stored in it.

eezdocker.sh migrate Migrates all of the custom configuration from the old SMU Docker container into the new container of a new SMU release.

eezdocker.sh reset Deletes the current SMU Docker container. If the **eezdocker.sh start** command is run later, a SMU Docker container is created.

Warning: All custom configuration is lost. Run this command only if you must reset to factory defaults.

eezdocker.sh uninstall

Deletes the current SMU Docker container and the SMU Docker image.

Warning: All custom configuration are lost. Run the command only if you must remove SMU from your Docker environment.



Container restart policies

After your Docker containers are set up and running in production, you must configure some Docker policies to restart containers after specific events, such as failures or if the zCX server is restarted for a planned change. Four restart policies are available that help you to automatically restart your containers when they exit.

In this section, we provide guidelines to change the container restart policy to ensure all of your containers are up and operational when the zCX server is restarted.

This chapter includes the following topics:

- ▶ 12.1, “Docker restart policies introduction” on page 248
- ▶ 12.2, “Summary” on page 251

12.1 Docker restart policies introduction

Containers are inherently ephemeral. They are routinely destroyed and rebuilt from a previously pushed application image. Another specific characteristic is the ability to be quickly stopped or ended that makes container easy to maintain. However, containers can be reloaded shortly after their termination in no-time, within milliseconds.

It is recommended that when running application containers in production, the restart policy is updated to ensure that the application is up and running after a zCX restart of an unexpected application failure.

Docker considers any containers to exit with a nonzero exit code to be crashed. By default, a crashed container remains stopped.

According to Docker documentation, the restart policies can be changed by supplying the **--restart** command line option:

```
docker run --restart=always --name <container_name> -d <image_name>
```

The options that are listed in Table 12-1 can be specified with the **--restart flag**, based on the required policy. These options also effect how the container starts.

Table 12-1 Restart flags

Flag	Description
no	Do not automatically restart the container (default).
on-failure	Restart the container if it exits because of an error, which manifests as a nonzero exit code.
always	Always restart the container if it stops. If it is manually stopped, it is restarted only when Docker daemon restarts or the container itself is manually restarted.
unless-stopped	Similar to always, except that when the container is stopped (manually or otherwise), it is not restarted even after Docker daemon restarts.

For the examples in this book, we supplied **--restart=always** so that a container always is restarted after the zCX instance is restarted. When a container is manually stopped by using the **docker stop <container_name>** command, the restart policy is not applied to the container.

Consider these options to better control container restarts on zCX.

In the following steps, we configure a Docker container on our zCX instance to demonstrate an example for the Docker restart policy and how it is important for production environments. We create a special container that displays a message and then exits with code 1 to simulate a crash:

1. Run the following commands to create a folder for the restart policy demonstration:

```
mkdir docker-restart-example
cd docker-restart-example
```

2. Create a file that is named `docker-crash.sh` with the following content:

```
#!/bin/bash
echo $(date) "Booting up..."
sleep 10
exit 1
```

This script displays a message, waits for 10 seconds, and then exits with code 1, which indicates an error.

3. Create the file `Dockerfile.policy` with the following content:

```
FROM ubuntu:latest
ADD docker-crash.sh /
CMD /bin/bash /docker-crash.sh
```

4. Build the new image by running the following command:

```
docker build -t docker-restart-example-img -f Dockerfile.policy .
```

You receive the following output:

```
Sending build context to Docker daemon 3.072kB
Step 1/3 : FROM ubuntu:latest
----> f8835575bd80
Step 2/3 : ADD docker-crash.sh /
----> 447fe066defa
Step 3/3 : CMD /bin/bash /docker-crash.sh
----> Running in 9f48429b6d5c
Removing intermediate container 9f48429b6d5c
----> cba8fc357912
Successfully built cba8fc357912
Successfully tagged docker-restart-example-img:latest
```

5. Start a container instance that is named `app-restart-default` by running the following command:

```
docker run -d --name app-restart-default docker-restart-example-img
```

6. Run the **docker ps** command to check the status of the `app-restart-default` container and confirm it exited and it is not running (see Example 12-1).

Example 12-1 Output of docker ps command

admin@31df3838c951:~\$ docker ps			
CONTAINER ID	IMAGE	COMMAND	CREATED
STATUS	PORTS		NAMES
a7503f1d5c08	jenkins-s390x	"/sbin/tini -- /usr/..."	24 hours ago
Up 24 hours	0.0.0.0:8080->8080/tcp, 0.0.0.0:50000->50000/tcp		jenkins-app
5bac28336b8b	sshd-img	"/usr/sbin/sshd -D"	26 hours ago
Up 26 hours	0.0.0.0:10000->10000/tcp, 0.0.0.0:32769->22/tcp		sshd-app
5e09f5fc924d	ubuntu	"/bin/bash"	36 hours ago
Up 36 hours			stupefied_newton
1e2b54213f2b	portainer/portainer:linux-s390x	"/portainer"	9 months ago
Up 4 minutes	0.0.0.0:9000->9000/tcp		portainer
31df3838c951	ibm_zcx_zos_cli_image	"sudo /usr/sbin/sshd..."	9 months ago
Up 36 hours	8022/tcp, 0.0.0.0:8022->22/tcp		ibm_zcx_zos_cli
844a59b71115	ibmcom/registry-s390x:2.6.2.3	"registry serve /etc..."	9 months ago
Up 36 hours	0.0.0.0:5000->5000/tcp		registry

7. Run the **docker logs app-restart-default** command to check the logs and confirm that the application was started but exited after 10 seconds. You should receive the following message:

```
Sat Jun 27 11:22:32 UTC 2020 Booting up...
```

By default, if an application that is running when a container crashes, the container stops and it remains stopped until it is restarted.

Depending on the application failure, restarting a failed process might correct the problem. Docker can automatically attempt to restart Docker a specific number of times before it stops trying.

In the following example, we create a container by using the **--restart=on-failure:3** option, where the number 3 indicates how many times Docker should try to restart before stopping the container:

```
docker run -d --name app-restart-3 --restart=on-failure:3
docker-restart-example-img
```

If you run the **docker ps** command, you see app-restart-3 container in the running container list. However, after some seconds, the container exits with a return code 1 and stops.

8. Run the **docker logs app-restart-3** command and check for boot messages, as shown in Example 12-2.

Example 12-2 Output of docker logs command

```
Sat Jun 27 11:33:49 UTC 2020 Booting up...
Sat Jun 27 11:34:05 UTC 2020 Booting up...
Sat Jun 27 11:34:25 UTC 2020 Booting up...
Sat Jun 27 11:34:40 UTC 2020 Booting up...
```

9. The following command uses the **--restart=always** option, in this case, Docker keeps trying until it is stopped. To demonstrate this option, run the following command:

```
docker run -d --name app-restart-always --restart=always
docker-restart-example-img
```

10. Wait a few moments and run the **docker logs app-restart-always** command many times to see the restart attempting by way of the logs, as shown in Example 12-3.

Example 12-3 Docker logs output for always restart flag

```
Sat Jun 27 11:44:15 UTC 2020 Booting up...
Sat Jun 27 11:44:33 UTC 2020 Booting up...
Sat Jun 27 11:44:49 UTC 2020 Booting up...
Sat Jun 27 11:45:08 UTC 2020 Booting up...
Sat Jun 27 11:45:22 UTC 2020 Booting up...
Sat Jun 27 11:45:38 UTC 2020 Booting up...
Sat Jun 27 11:45:53 UTC 2020 Booting up...
Sat Jun 27 11:46:10 UTC 2020 Booting up...
Sat Jun 27 11:46:27 UTC 2020 Booting up...
Sat Jun 27 11:46:45 UTC 2020 Booting up...
Sat Jun 27 11:47:00 UTC 2020 Booting up...
Sat Jun 27 11:47:26 UTC 2020 Booting up...
Sat Jun 27 11:47:43 UTC 2020 Booting up...
```

11. Do not forget to delete the example container by running the following command:

```
docker rm --force app-restart-always
```

12.2 Summary

As described in this chapter, four restart options are available to change the behavior of how Docker handles a container failure or unforeseen conditions. The use of an adequate policy can assure your container is up and running in your environment, even after a planned zCX outage.

Review each option and select the one that best applies to your application environment.

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



Additional material

This chapter refers to additional material that can be downloaded from the internet as described in the following sections.

Locating the web material

The web material that is associated with this publication is available in softcopy from the IBM Redbooks web server:

<ftp://www.redbooks.ibm.com/redbooks/SG248471/>

Alternatively, you can go to the IBM Redbooks website:

ibm.com/redbooks

Search for SG248471, select the title, and then, click **Additional materials** to open the directory that corresponds with the IBM Redbooks form number, SG248471.

Using the web material

The additional web material that accompanies this book includes the following files:

<i>File name</i>	<i>Description</i>
------------------	--------------------

DevOps_add_mat.zip	Contains folders listed as Agent, Gitea, and hello-node-app_files. Each folder contains sample images to use as you follow along in this book.
---------------------------	--

CicsToKafkaProjects.zip	
--------------------------------	--

	Contains folders listed as CatalogEvents, CicsToKafkaBunde, and CicsToKafkaDemo. Each folder contains
--	---

Downloading and extracting the web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material .zip file into this folder.

Related publications

The publications that are listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this bookbook.

IBM Redbooks

The *Getting started with z/OS Container Extensions and Docker*, SG24-8457 IBM Redbooks publication provides more information about the topics in this document. Note that this publication might be available in softcopy only.

You can search for, view, download, or order this book and other Redbooks, Redpapers, Web Docs, draft, and additional materials, at the following website:

ibm.com/redbooks

Online resources

The following web pages are also relevant as further information sources:

- ▶ <https://developer.ibm.com/components/kafka/articles/an-introduction-to-apache-kafka/>
- ▶ <https://ibm-cloud-architecture.github.io/refarch-eda/kafka/readme/>
- ▶ https://www.ibm.com/support/knowledgecenter/SSZJPZ_11.5.0/com.ibm.swg.im.iis.event.doc/topics/t_using_Apache_Kafka.html
- ▶ https://zookeeper.apache.org/doc/current/zookeeperStarted.html#sc_RunningReplicatedZooKeeper
- ▶ <https://www.digitalocean.com/community/tutorials/how-to-install-and-configure-an-apache-zookeeper-cluster-on-ubuntu-18-04>

Help from IBM

IBM Support and downloads:

ibm.com/support

IBM Global Services:

ibm.com/services

Redbooks

IBM z/OS Container Extensions (zCX) Use Cases

SG24-8471-00

ISBN 0738459119



(0.5" spine)

0.475" <-> 0.873"

250 <-> 459 pages



SG24-8471-00

ISBN 0738459119

Printed in U.S.A.

Get connected

