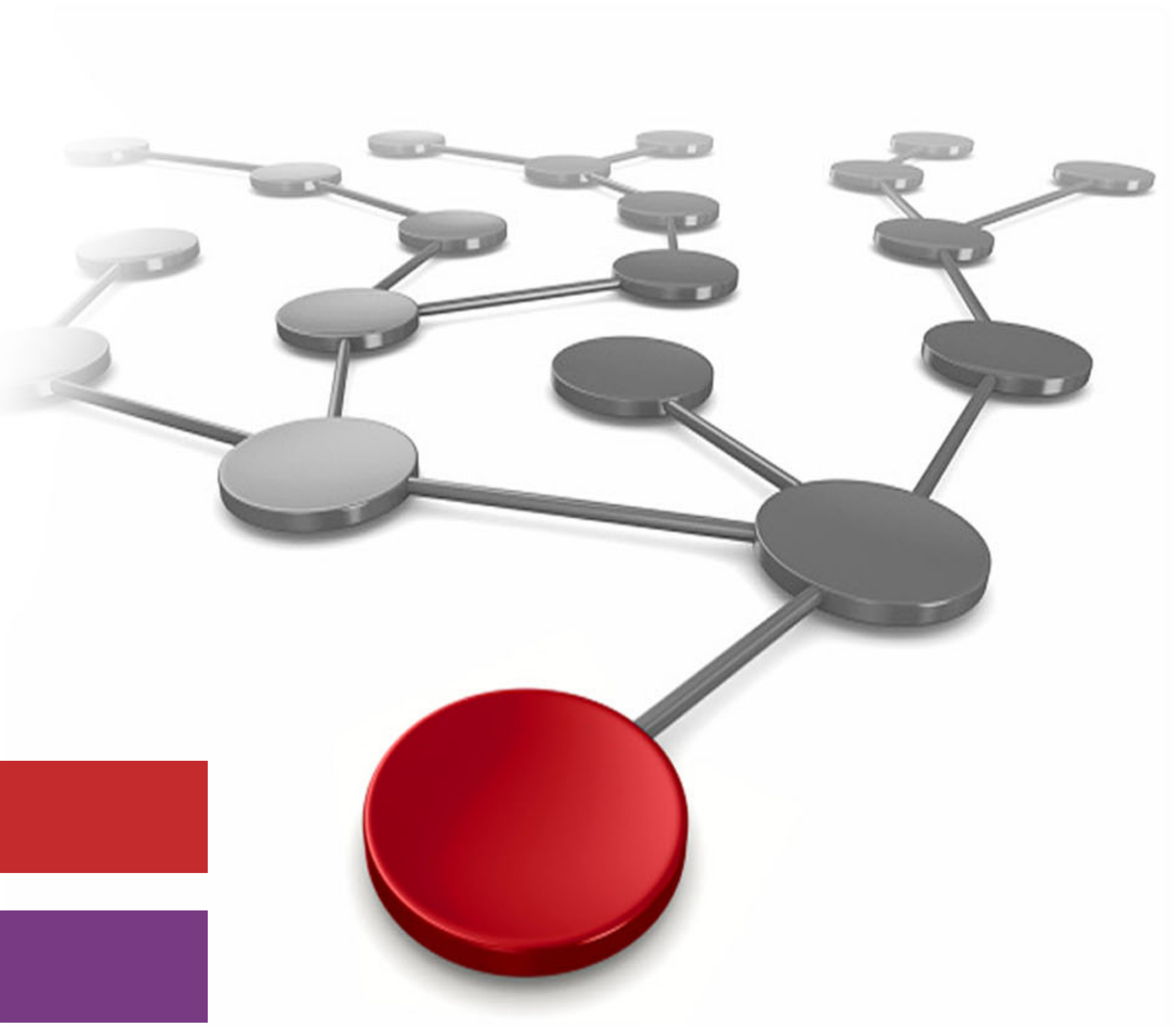
Redbooks
ibm.com/redbooks

How to Use IBM Cloud Object Storage When Building and Running Cloud Native Applications

Giri Badanahatti
Aaron Thomas Binford
Charlie Crawford
Daniel Fitzgerald
Vasfi Gucer
Wesley Leggette
Mrudula Madiraju
Daniel Pittner
Joeri Van Speybroek



Cloud

Storage

IBM

Redpaper

International Technical Support Organization

How to Use IBM Cloud Object Storage When Building and Running Cloud Native Applications

August 2018

REDP-5491-00

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

First Edition (August 2018)

This edition applies to IBM Cloud Object Storage Version 1.0.

This document was created or updated on November 15, 2018.

Contents

Notices

Trademarks

Preface

This IBM® Redpaper™ publication presents a series of tutorials for cloud native developers just getting started with IBM Cloud™ and IBM Cloud Object Storage. Within the context of a car insurance application, this paper presents an introductory series of linked modules that allow developers unfamiliar with either IBM Cloud or cloud native development to get started with application development using IBM starter kits. This allows you to become familiar with the types of services available on IBM Cloud, and to develop a sense of which patterns and choices are appropriate for different use cases.

Some of the technologies and products covered in this book are Cloudant®, Watson™ Analytics, machine learning, elastic search, Kubernetes, containers, pre-signed URLs, Aspera®, and SQL Query. In addition to the technical integration steps, it also presents a business case for integrating these technologies and products with IBM Cloud Object Storage.

The target audience for this paper is cloud native developers and cloud object storage specialists.

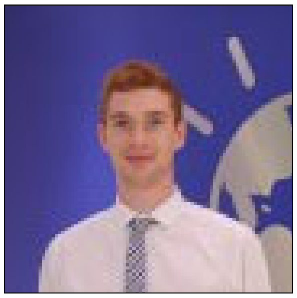
Authors

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

Badanahatti Giri is a System Architect within the IBM Cloud Object Storage Alliance Integration Team. He has around 20 years of technology experience split between Telecommunications and Storage. He is passionate about finding new solutions and integrations with IBM Cloud Storage.

Aaron Thomas Binford is a System Architect with the IBM Cloud Object Storage Alliance Integration Team. He has 15 years of experience, including Information Technology architecture, DevOps engineering, and concurrent/distributed systems development.

Charlie Crawford is a Senior Software Developer within the Data Services Center of Competency team. He has over 18 years of experience, ranging from quality assurance, development, Watson AI, and IBM Cloud.



Daniel Fitzgerald is a sales engineer working for IBM Cloud with a wide range of experience across many industries and a background in software development. Daniel helps clients to build Cloud native apps and employ DevOps practices and tools across their software development lifecycle. In his previous role, he worked as a DevOps engineer on the Customer Information Control Systems (CICS®) product at the IBM Hursley development lab.

Vasfi Gucer is an IBM Technical Content Services Project Leader with the Digital Services Group. He has more than 20 years of experience in the areas of systems management, networking hardware, and software. He writes extensively and teaches IBM classes worldwide about IBM products. His focus has been primarily on cloud computing, including cloud storage technologies for the last six years. Vasfi is also an IBM Certified Senior IT Specialist, Project Management Professional (PMP), IT Infrastructure Library (ITIL) V2 Manager, and ITIL V3 Expert.

Wesley Leggette is responsible for Architecture and Development Enablement for IBM Cloud Object Storage. He is a Master Inventor and has over 500 issued and pending patents relating to dispersed and cloud storage. He previously worked at Cleversafe®, Inc. until 2006 when the Chicago-based startup was acquired to form the basis of the IBM Cloud Storage offering. He plays a key role in integrating IBM Cloud Object Storage into the IBM Cloud experience.

Mrudula Madiraju is an Advisory Software Engineer in IBM India. She has a lot of experience in global services delivery model for customer projects, on-premises big data product development, and big data cloud platform development. In her most recent role, she works as a customer consultant with focus on working with key focus on S3 object store. Previously she worked as the Senior TechLead in the Cluster Provisioning Dev team on Hadoop & Spark Service on the Cloud.

Daniel Pittner is a software architect working for IBM Cloud SQL query with a strong focus on cloud native engineering and DevOps. He has over 10 years of experience as a full-stack engineer and architect building software and delivery automation ranging from cloud native services over enterprise content management systems to unstructured archiving solutions. In a previous role, he was an architect for Box Relay, a joint offering between Box and IBM.

Joeri Van Speybroek is a Cloud Storage Architect within the EU IBM Cloud Object Storage Technical Sales team. As a storage enthusiast, he likes to pursue the evolution of enterprise storage and help clients to enhance their storage architectures. He has experience in block, file, and object storage architectures where he tries to provide the best fit for any environment/application in an on-premises, off-premises, and hybrid design. His main interest is to detect/create object storage use-cases to enable/convince clients to offload their installed systems to the IBM Cloud Object Storage platform.

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

Ernest Keenan
IBM Technical Content Services, Austin Center

Dawn McKenna, Laura Noonan, Michael Factor, Michael J Fork, Mike Lamb, Naeem Altaf, Ramin Rouzbeh, Riz Amanuddin
IBM US

Torsten Steinbach
IBM Germany

Now you can become a published author, too!

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

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

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

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

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

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

- Mail your comments to:

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

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

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

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

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

- Stay current on recent Redbooks publications with RSS Feeds:

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

1

IBM Cloud Object Storage primer

This series of tutorials explains how to build a cloud-native web application on IBM Cloud. You will be building a consumer-facing Insurance application. This application will use IBM Cloud Object Storage, and will also employ AI and machine learning to provide a unique digital experience for a customer.

This module explains what object storage is, and how and why you should use it.

Note: These modules build on each other, so you should complete all of them in the order they are presented.

This chapter includes the following sections:

- Learning objectives
- Getting started
- What is object storage?
- Provisioning choices for IBM Cloud Object Storage
- API compatibility
- Security and encryption
- Data ingest and access options
- Next steps
- Other references

1.1 Learning objectives

After completing this module, you will be able to:

- Describe what object storage is.
- Understand when to use object storage, file system, and databases in a cloud native application.
- Understand the location and resiliency options available in IBM Cloud Object Storage, and what the tradeoffs are for each resilience option.
- Describe the available IBM Cloud Object Storage tiers.
- Consider IBM Cloud Object Storage security options, including authentication, access control, and encryption.
- Understand various data ingest options.

1.2 Getting started

Before you begin, sign up for a free IBM Cloud account or if you already have one, sign in. Note that to leverage the full potential of the sample application that will be presented in this Tutorial series, you need an IBM Cloud Pay-As-You-Go account. When creating or upgrading a lite account to a Pay-As-You-Go, IBM Cloud offers a USD 200 credit that can go towards creating, testing, and enhancing this application.

1.3 What is object storage?

Object storage is a way of organizing large amounts of data (objects) into collections (buckets) by giving each object an identifying name (object key). Because buckets are uniquely named, the combination of the bucket name and object key form a Uniform Resource Identifier (URI). A useful analogy is valet parking: You do not need to know the space number where the car is parked. You just provide a ticket to the valet and you get your car back. With object storage, you do not need to know which physical disks or partitions data is written to and read from. If you know the URI, the system can find your data.

IBM Cloud Object Storage provides a REST-based API that allows applications to directly access data both within IBM Cloud data centers and over the internet. Object storage is ideal for unstructured data, such as images and video.

Object storage can also be used for the storage and query of structured data using Spark, Hadoop, and IBM SQL Query, as shown in Figure 1-1. This concept is explored in Modules 6, 8, and 9.

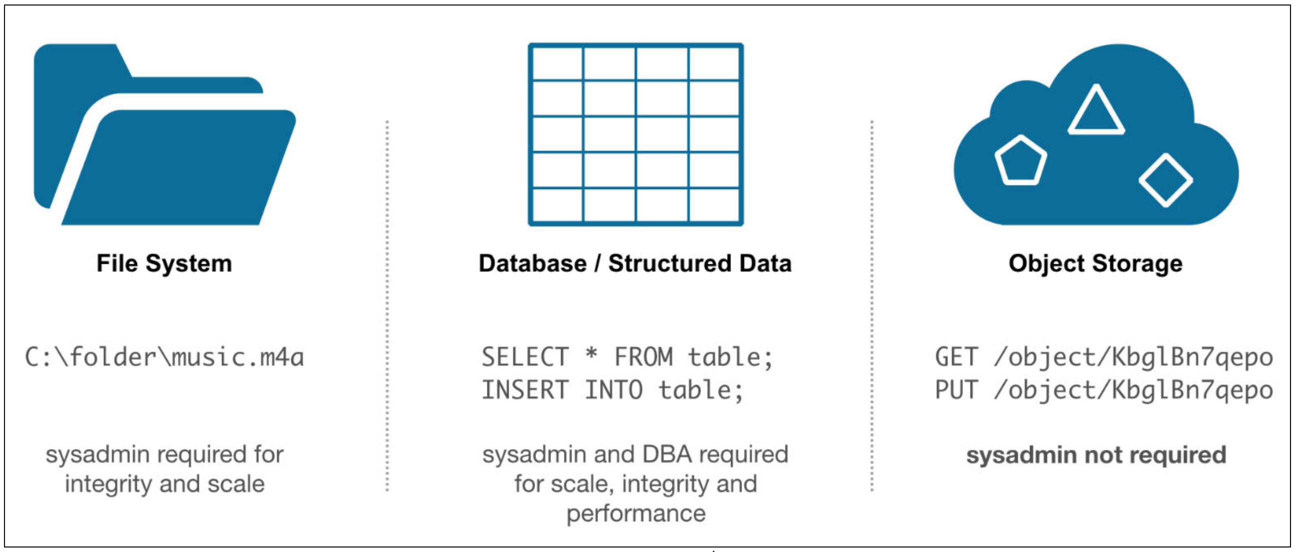


Figure 1-1 Object Storage comparison with other storage options[1]

Unlike a file system or block storage, object storage allows fully atomic writes and overwrites, not partial modification. This behavior is closer to a database and greatly simplifies application development.

IBM Cloud Object Storage also provides several multi-AZ regional and even cross-regional resiliency options. Your application can be deployed redundantly across many locations with data following along automatically. There is no need for your DevOps team to manage durability or availability.

File systems are still useful for cloud native development. This series uses ephemeral file storage as a staging area to process images using a library that requires file storage. By only using the file system as a temporary storage place, you can avoid having to implement complex file system synchronization technologies. If you store permanent data in either a database or object storage, the application's data is automatically available wherever instances of the application are deployed.

Unlike a database, object storage does not provide multiple indexes or multi-row and multi-table transactions. Databases also provide much higher transactional throughput and can process queries with lower average latencies. However, object storage makes up for these limitations with scalability and cost. There are no practical limits to the number of objects that can be stored in a Cloud Object Storage bucket, and object sizes can grow as large as 10 TB. Per GB per month, object storage can be orders of magnitude cheaper than databases.

A database can also provide more flexibility transactional capabilities. An ACID-compliant database can simplify creating a correct application. However, it is worth noting that not all databases provide transactional capabilities, and there is often good reason to choose one that does not. In this series, we select IBM Cloudant, based on the open source Couch DB, as our database.

[1] Source: `https://www.ibm.com/blogs/cloud-computing/2017/02/01/object-storage-benefits-myths-and-options/`

The Cloudant horizontal scaling architecture provides many of the same scalability and multi-DC resiliency properties as object storage. This feature makes it a good fit for a scale-out application that has instances running across the globe. However, it has a loose eventual consistency model that does put some burden on your application to ensure correct implementation. The cost per GB of Cloudant is also much lower than other databases, but still much higher than IBM Cloud Object Storage.

It is worth noting that IBM Cloud Object Storage provides a strong consistency model. It provides read-after-write consistency for all updates in all regions. IBM provides the only object storage system with immediate strong consistency for cross-region buckets (stored in multiple geographic regions).

1.4 Provisioning choices for IBM Cloud Object Storage

Information stored with IBM Cloud Object Storage is encrypted and dispersed across multiple geographic locations. This service uses the distributed storage technologies provided by the IBM Cloud Object Storage System™ (formerly Cleversafe).

Like other IBM Cloud services, IBM Cloud Object Storage can be provisioned by creating an IBM Cloud Object Storage *service instance*. A service instance is an administrative unit that can be used for access control and for billing separation. One or more *buckets* can be created within a service instance. Buckets are created in a specific *region*, which determines the physical location and resiliency of objects that are stored in that bucket.

IBM Cloud Object Storage is available with three types of resiliency: Cross Region, Regional, and Single Data Center:

- **Cross Region** provides higher durability and availability than using a single region at the cost of slightly higher latency. This type is currently available in the US and EU.
- **Regional** service reverses those tradeoffs and distributes objects across multiple availability zones within a single region. This type is available in several regions across the globe. If a region or Availability Zone is unavailable, the object store continues to function without impediment.
- **Single Data Center** distributes objects across multiple machines within the same physical location.

A full list of the current object storage locations and resiliency options can be found in the Regions and Endpoints documentation. Figure 1-2 shows the tiers for Storage Class.

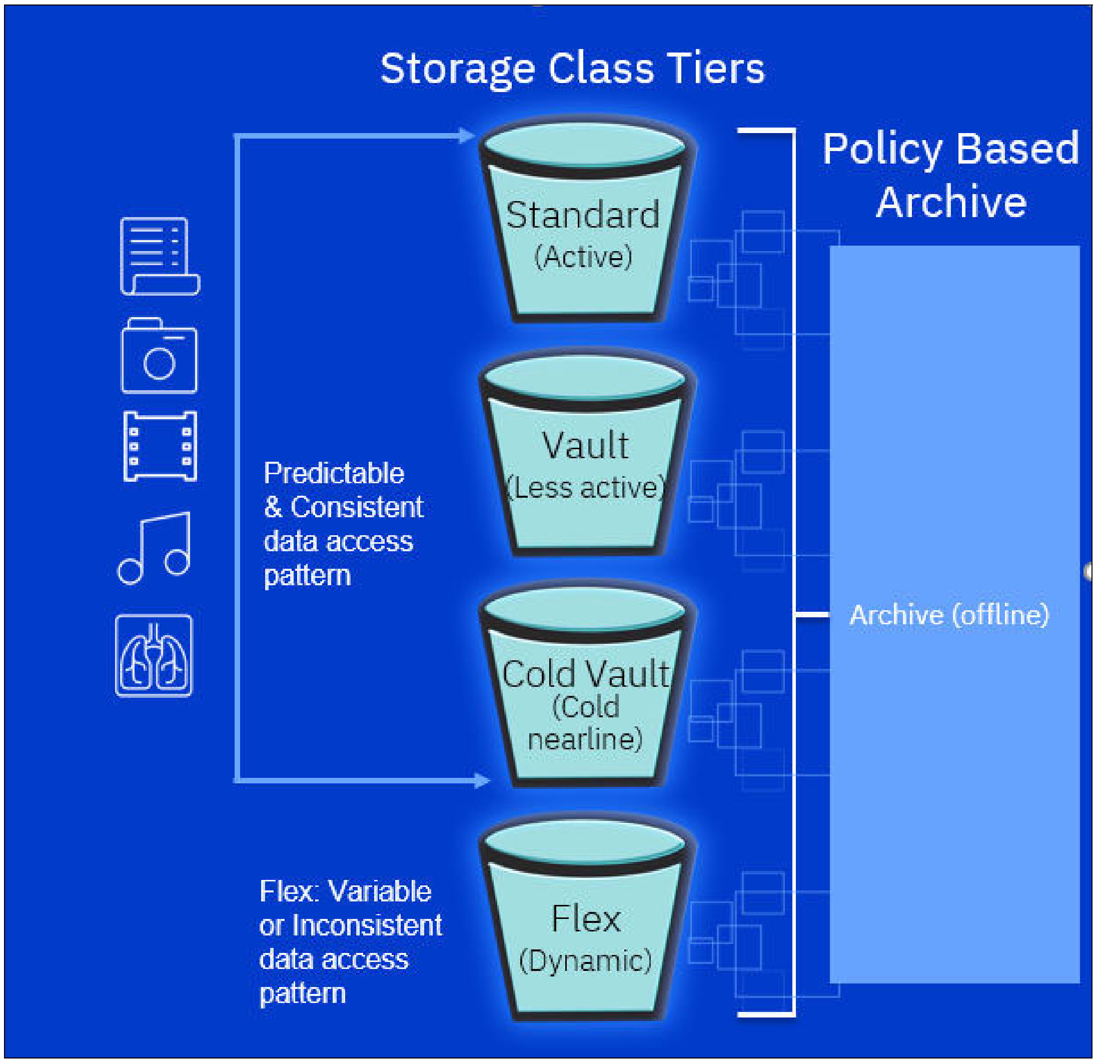

Storage Class Tiers
Standard (Active)
Vault (Less active)
Cold Vault (Cold nearline)
Flex (Dynamic)
Predictable & Consistent data access pattern
Flex: Variable or Inconsistent data access pattern
Policy Based Archive
Archive (offline)

Figure 1-2 Storage Class tiers

IBM Cloud Object Storage provides several storage class tiers to allow a DevOps team to optimize costs:

- Standard: For active data that is accessed multiple times a month
- Vault: For less active data that is accessed once a month or less
- Cold Vault: For data accessed several times a year, and needs immediate real-time access
- Flex: For dynamic data, access frequency varies monthly or for mixed “hot” and “cold” workloads

IBM Cloud Object Storage also provides an archive policy that can be applied for deep auxiliary storage for seldom accessed data. It is even more cost effective to store data in the archive tier, but access times for this archived data can be up to 15 hours. Your application must be archive aware and must be able to manage restoring objects before access to effectively use this policy. This series does not examine using archive policy buckets in detail, but it is worth considering building this capability into your application when applicable.

1.5 API compatibility

Most third-party applications that work against AWS S3 can be used against IBM Cloud Object Storage. In most cases, this feature allows you to migrate your data to the IBM Cloud using an application.

If you are a developer who is modifying an application written for Amazon S3 to work against IBM Cloud Object Storage, most S3 API functions remain the same. Additionally, the IBM Cloud Object Storage software development kit (SDK) is available in Java, Python, and Node-js. For many applications, the Cloud Object Storage SDK is a drop-in replacement for the AWS SDK, making it easy for developers to migrate their applications and support Cloud Object Storage API-specific features.

In this series, we chose JavaScript and Node.js for development. We also use the IBM Cloud Object Storage Node.js SDK.

For both third-party applications and migrated applications, note the following differences when provisioning storage and setting up authentication and access control:

- Provisioning: A Bluemix® service instance must be created before buckets can be created. See Module 2 for more details about how to do this.
- Authentication: For maximum capability, users can provision Cloud Object Storage HMAC Keys within a service instance. Configuring an application with HMAC keys allows it to use AWS V4 signature authentication unmodified. However, a fully migrated application should provide support for OAuth2 API Keys for maximum compatibility.
- Location Constraint: The IBM Cloud regions are different from Amazon AWS regions, and have different names. The location constraint field also needs to include a storage class for the bucket and a region code if a storage class other than "standard" is being used.
- Endpoints: The IBM Cloud Object Storage regional and cross region endpoints are different from Amazon AWS endpoints. Administrators must modify the connection endpoint in the application to point to a specific IBM Cloud Object Storage endpoint, depending on where they would like to create a bucket.
- AWS IAM policies and S3 bucket policies: Administrators need to change to IBM Cloud IAM policies. IBM Cloud IAM policies support both user access control as well as cross-service and cross-account access control.
- S3 ACLs: S3 ACLs support both Cross-Account Access Control and Anonymous Access:
  - Cross-Account Access Control: Administrators need to use IBM Cloud IAM policies to provide cross-account access control. Applications that manage cross-account access control and require ACL support to do this must be modified.
  - Anonymous Access: Applications that use ACLs to provide anonymous access to buckets and objects do not need to be modified.
- Versioning: Older versions of an object will not be retained when an application writes new data. Applications will not be able to list retained object versions.

1.6 Security and encryption

Like other services in IBM Cloud, IBM Cloud Object Storage is secured using IBM Identity and Access Management (IAM). IBM Cloud Object Storage offers integrated support for policies and permissions. Your DevOps team can set bucket level permissions for users and your applications. IBM Starter Kits make it easy to configure your application to access your IBM Cloud Object Storage bucket, as shown in Figure 1-3.

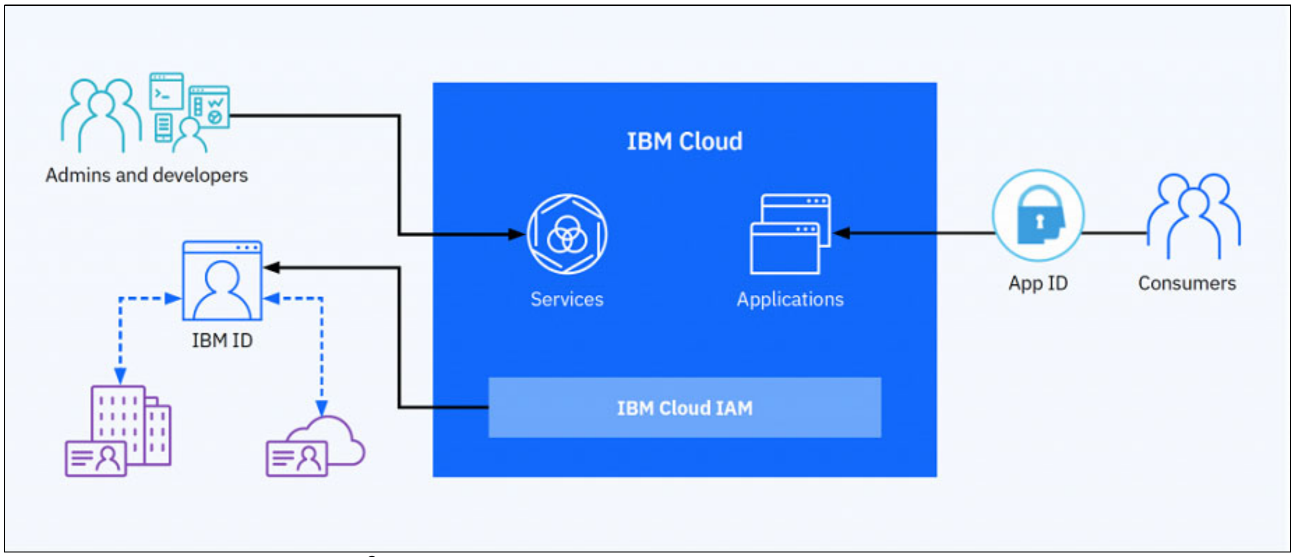


Figure 1-3 Security configuration[2]

IBM Cloud Object Storage supports both IBM Cloud's native API keys and OAuth2 authentication mechanism, as well as supporting HMAC authentication for third-party applications and SDKs.

This series also covers user authentication with IBM App ID. It is worth noting the difference between these two mechanisms. Both use OAuth2 authentication flows. IAM manages access control between IBM Cloud services, and between your application and the IBM Cloud services it uses. App ID manages authentication and identity between your application and users.

See Getting started with IAM and Using HMAC credentials in the IBM Cloud Object Storage documentation for more information about authentication and access control mechanisms.

[2] Source: https://www.ibm.com/blogs/bluemix/2018/02/five-fundamentals-cloud-security/

Figure 1-4 shows an example configuration of encryption across a network.

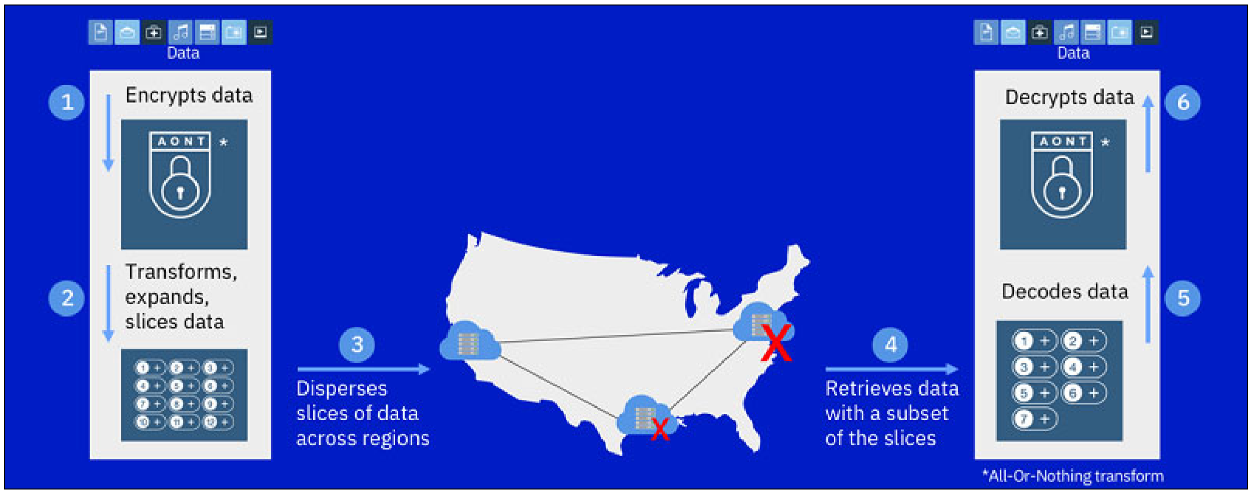

Data
1
Encrypts data
AONT *
2
Transforms, expands, slices data
3
Disperses slices of data across regions
4
Retrieves data with a subset of the slices
Data
Decrypts data
6
AONT *
Decodes data
5
*All-Or-Nothing transform

Figure 1-4 Encryption across a network

IBM Cloud Object Storage has several options for data encryption.

First, all data is encrypted in transit using TLS 1.2, both between data centers and within data centers. You should also configure your application to use HTTPS when communicating with IBM Cloud Object Storage and to your users.

All data stored at rest is encrypted using an encryption mechanism called “SecureSlice”, which is all-or-nothing encryption. This mechanism encrypts data using a per-segment (a subset of an object) key before each object segment is erasure coded and stored across multiple storage nodes across multiple data centers. The exception to this rule is buckets that are stored in Single Data Center locations. It is not possible to decrypt the data unless a threshold of slices is recovered.

Additionally, if you would like to manage the keys used for encryption, you have several options to further enhance security:

- Client-side encryption can be used to encrypt data before it is sent to the storage server.
- Server-side Encryption with Customer-Provided Keys (SSE-C) support allows your application to provide key material on a per-object basis, allowing IBM Cloud Object Storage to perform the cryptography. When using SSE-C, IBM Cloud Object Storage only holds the key material in memory, discarding it as soon as the operation is complete.
- Server-side Encryption with IBM Key Protect (SSE-KP) support allows you or a security officer to upload or generate key material on IBM Key Protect, a centralized key management system (KMS) backed by secure FIPS-140-2 Level 2 Hardware Security Modules. When using SSE-KP, a bucket is linked to an encryption key in Key Protect, with access controlled from a central dashboard. Unlike SSE-C, besides during initial bucket setup, SSE-KP is transparent to your application.

See Manage Encryption in the IBM Cloud Object Storage documentation for more information about these options.

1.7 Data ingest and access options

Additionally, applications must be designed to use the most efficient method for transferring data. When uploading a single object at a time, there are two broad mechanisms for both upload and download of data.

For writing data (performing a PUT operation using the REST API):

- ▶ Objects can be written either as a single part
- ▶ Broken up into multiple, smaller chunks of data using multipart upload

Likewise, for reading data (performing a GET operation using the REST API):

- ▶ Objects can be read as a single part
- ▶ The HTTP Range header can be used to read specific offsets.

Additionally, the IBM Cloud Object Storage SDKs provide a utility to manage uploading one or more objects at a time. This utility is called `ManagedUpload` or `TransferManager`, depending on the language being used. The `ManagedUpload` utility can be used to manage transfer of many objects at once, automatically manage multi-part uploads for files, handle retries during failures, and provide upload or download transfer status using a callback mechanism. This utility can make transferring of large amounts of data much easier.

More information about using these various methods in your application is available in Chapter 7, “Advanced Object Storage integration patterns” on page 83.

Additionally, objects can be managed in the IBM Cloud Object Storage console. The UI allows you to list, upload, download, and view additional information about objects, as well as manage buckets and service instances. By default, uploading and downloading objects from object storage simply uses your built-in web browser download and upload function. This technique is sufficient for casual uploads of a few files.

IBM Cloud Object Storage also provides built-in integration with Aspera, a high-speed data transfer service using an optimized transfer protocol that increases throughput in high latency and packet loss environments. This service is automatically available in the IBM Cloud Object Storage console, and does not require custom configuration or service provisioning. You can use Aspera to upload and download data by downloading a browser plug-in.

Aspera is free of charge for ingest, but there is a fee when using Aspera for reading data.

Consider using Aspera in situations where you need to upload large amounts of data. You see an example of this approach in Chapter 8, “Discover insights using Watson Services” on page 113 and Chapter 9, “Performing more advanced functions with machine learning” on page 151, which cover uploading large amounts of training data for storage.

For more information about upload options using the UI and a list of regions where Aspera is available, see uploading data in the IBM Cloud Object Storage documentation.

1.8 Next steps

The next module provides a broader overview of IBM Cloud and provides instructions for setting up your development environment.

This series continues to explore various aspects of IBM Cloud Object Storage, and how to use it effectively in a cloud-native application:

- Chapter 4, "Scaffolding using a Starter Kit" on page 29: Explore how IBM Cloud Object Storage integrates with cloud native development and starter kits.
- Chapter 5, "Storing metadata in Cloudant" on page 51: Learn about how IBM Cloud Object Storage is an ideal repository for database backups.
- Chapter 6, "Using IBM Cloud Object Storage" on page 63: Learn how to order IBM Cloud Object Storage and how to integrate it into your application as the primary storage for user uploaded data.
- Chapter 7, "Advanced Object Storage integration patterns" on page 83: Explore some of the advanced integration patterns mentioned in this module.
- Chapter 8, "Discover insights using Watson Services" on page 113 and Chapter 9, "Performing more advanced functions with machine learning" on page 151: Using Watson Visual Recognition and other machine learning tools, learn how to use Object Storage as a low cost and easy to use part of the machine learning workflow.

1.9 Other references

- IBM Cloud Object Storage docs

  https://console.bluemix.net/docs/services/cloud-object-storage/about-cos.html#about-ibm-cloud-object-storage
- IBM Cloud Object Storage API Reference

  https://console.bluemix.net/docs/services/cloud-object-storage/api-reference/about-api.html#about-the-ibm-cloud-object-storage-api
- IBM Cloud Object Storage Java SDK

  https://console.bluemix.net/docs/services/cloud-object-storage/libraries/java.html#using-java
- IBM Cloud Object Storage Python SDK

  https://console.bluemix.net/docs/services/cloud-object-storage/libraries/python.html#using-python
- IBM Cloud Object Storage Node.js SDK

  https://console.bluemix.net/docs/services/cloud-object-storage/libraries/node.html#using-node-js
- IBM Key Protect docs

  https://console.bluemix.net/catalog/services/key-protect
- IBM Aspera high speed data transfer product information

  https://www.ibm.com/cloud/high-speed-data-transfer

2

IBM Cloud primer

Welcome to part two in this tutorial series! This series explains how to build a cloud-native, consumer-facing Insurance application on IBM Cloud. This application is built according to best practices, uses IBM Cloud Object Storage at its core, and employs AI and machine learning to provide a unique digital experience for a customer.

This module explains what the IBM Cloud is, and how and why you would use it. IBM Cloud is a broad platform with many capabilities that are impossible to cover in such a short time. For this reason, this paper includes plenty of references and additional reading should you be interested in researching a topic in more depth.

This module contains the following sections:

- Learning objectives
- Getting started
- What is IBM Cloud?
- Best practices and documentation
- Next steps
- Other references

2.1 Learning objectives

After completing this module, you will be able to:

- ► Describe the components that make up the IBM Cloud.
- ► Choose the correct compute models for your use case.
- ► Describe the services that are available in IBM Cloud.
- ► Use the IBM Cloud command line interface to administer your account.

2.2 Getting started

Before you begin, sign up for a free IBM Cloud account, or, if you already have one, sign in. Note that to leverage the full potential of the sample application that will be presented in this Tutorial series, you need an IBM Cloud Pay-As-You-Go account. When creating or upgrading a lite account to a Pay-As-You-Go, IBM Cloud offers a USD 200 credit that can go towards creating, testing, and enhancing this application.

2.3 What is IBM Cloud?

IBM Cloud is a platform that combines Platform-as-a-Service (PaaS) and Infrastructure-as-a-Service (IaaS) with a rich catalog of services that can be easily integrated with your data to rapidly build applications.

Developers can use IBM Cloud to access a wide range of services and run times from the open source community, IBM, and third parties to build apps using a polyglot programming approach, as shown in Figure 2-1.

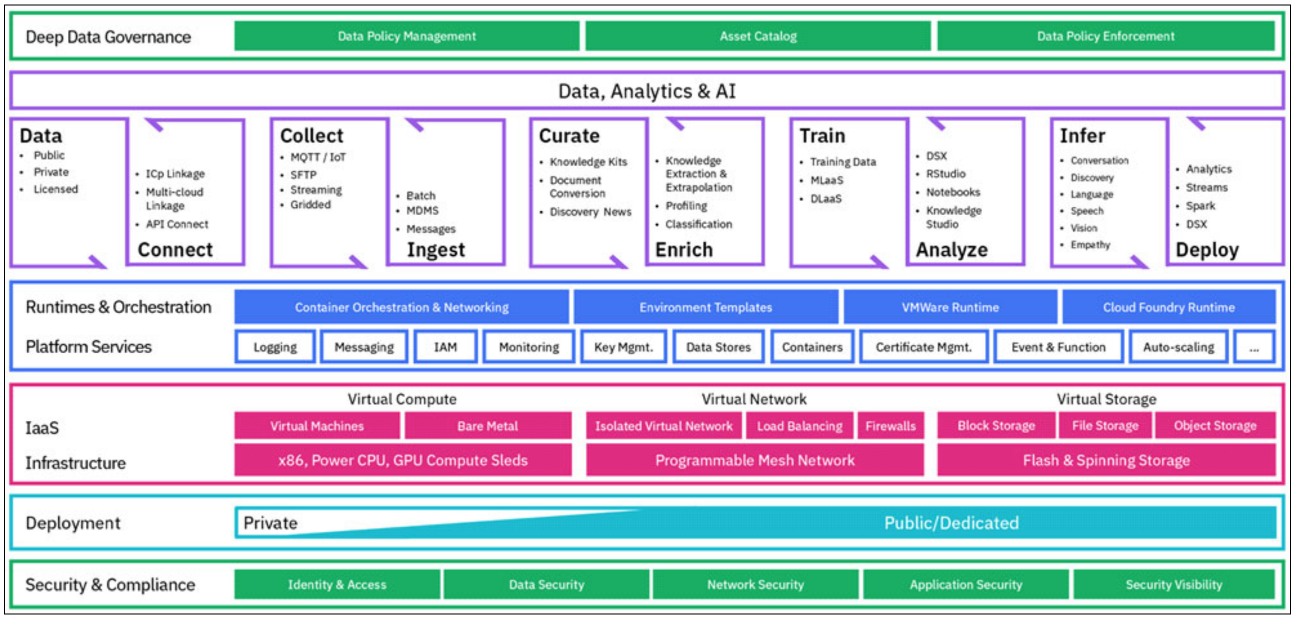


Figure 2-1 Available services

With IBM Cloud, you no longer need to make large investments in hardware and software to take an idea from the drawing board to a globally distributed production environment. This system can include compute, network, and storage infrastructure; open source platform services and containers; and software tools and services from IBM.

2.3.1 Running apps

IBM Cloud provides various ways for you to run your apps:

- Writing a web or mobile app? Choose Cloud Foundry, the immensely popular open source app platform to help you seamlessly deploy and scale your code.
- Need more flexibility over your apps deployment and want to use industry-standard tools like Docker and Kubernetes? Use the IBM Cloud Container Service, a container-based platform hosted on IBM Cloud, to run your Docker containers.
- Do you want to write fine-grained services that run in response to events and only charge you for the time they are being run, all without having to manage any infrastructure? Choose IBM Cloud Functions.

Developers often combine all of these platforms to help them create amazing things.

IBM Cloud is not just for cloud-native workloads. Virtual and bare metal servers are also available when you need more flexibility and control over your infrastructure or network. These servers work well for scenarios where you want to move an existing application to the cloud or deploy a piece of off-the-shelf software. Figure 2-2 shows the available options in IBM Cloud to run your applications.

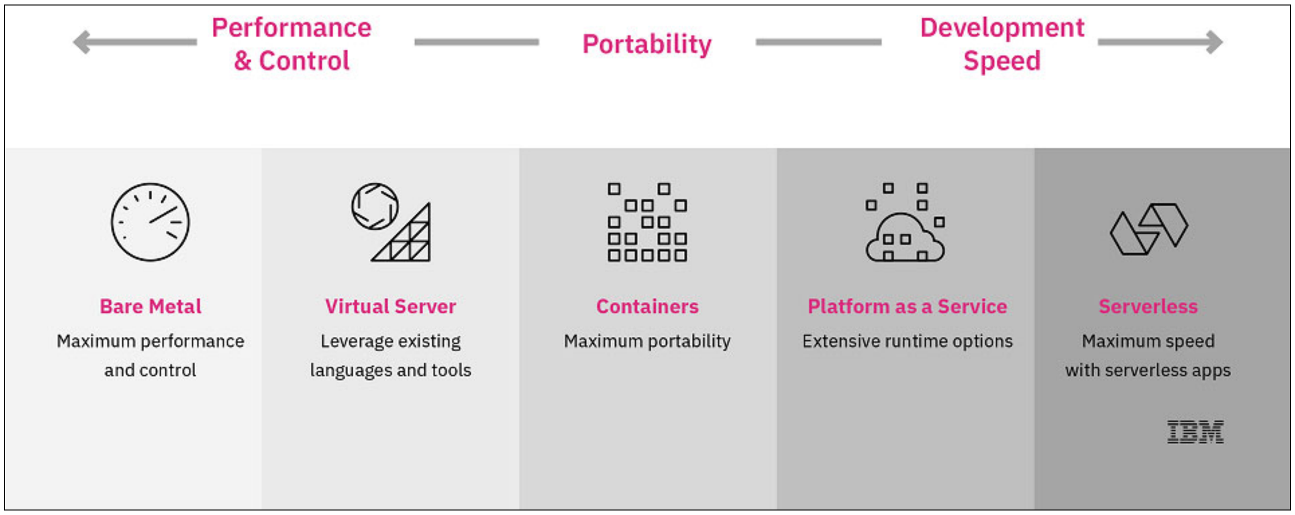


Figure 2-2 Available options in IBM Cloud to run your applications

2.3.2 Services

You can access a catalog of 180+ services from the open source community, IBM, and third parties such as New Relic, SendGrid, and Twilio. You can use these services as the building blocks of your app to get to market faster at a lower cost. Most come “as-a-service,” meaning that you do not need to manage infrastructure or be an expert in configuring a service to use it in your application.

- Application services: Access message queues, maps, VoIP, content management, email-as-a-service, and so on.
- Blockchain: Simplify developing, governing, and operating an IBM Blockchain network with Hyperledger fabric hosted on the IBM Cloud.
- Data and Analytics: Use fully managed SQL, NoSQL, big data, caching, and streaming. Includes open source applications from Spark, Hadoop, Mongo, Redis, and IBM such as DB2® and Cloudant.
- DevOps: Provides alerts, monitoring, CI/CD, pipelines, logging, and third-party services such as New Relic, Load Impact, and BlazeMeter. You can also use your own DevOps toolchains and integrate with your APIs.
- Infrastructure: Provision virtual machines and bare metal servers, application and global load balancers, subnets and VLANs, WAF and DDoS, and Content Distribution Network.
- Storage: Set up object, block, and file storage.
- Security: Manage encryption keys, certificates, capture audit trails and logs, and secure your apps against an enterprise directory.
- Watson: Build chatbots; extract insight from images, text, voice, and video; translate languages; analyze sentiment; build your own custom models; and use Deep-Learning-as-a-Service.

2.3.3 Build smarter apps that employ Artificial Intelligence, Machine Learning, and Deep Learning

Watson is a collection of services that employ AI, ML, and DL techniques to extract insight from language, speech, vision, and data. You can combine these services together to build applications that understand, reason, learn, and interact with humans through unstructured data like imagery and language, just like humans do. Figure 2-3 shows the available services from IBM Watson®.

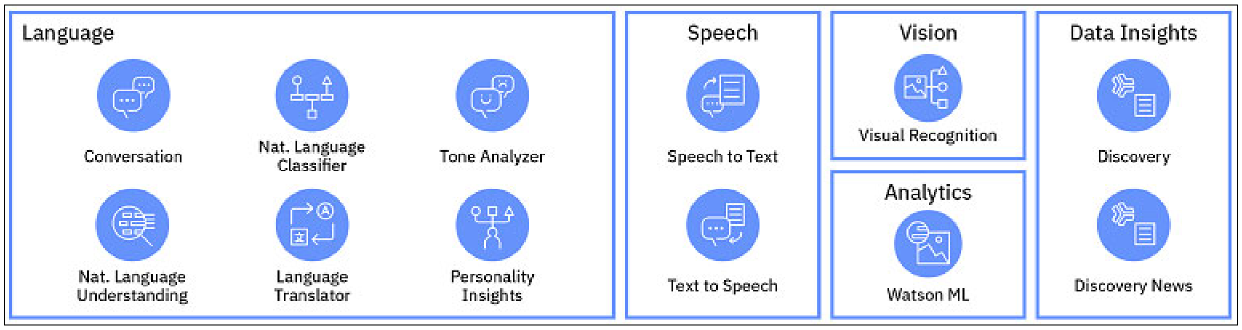


Figure 2-3 IBM Watson features

Each of these services provides an API to interact with so you can incorporate them into your apps. Many of these services also provide tools such as web interfaces to help you build and train models. For example, Watson studio gives you tools to train custom visual recognition classifiers, and Conversation provides an interface to create dialog trees that will become a digital assistant.

One of the stated goals of IBM Watson is to democratize AI and make it available to developers who have not studied statistics or learned a language like Python or Scala. IBM also recognizes that in many cases, the professionals who have the domain expertise to train a machine learning model are not computer scientists. For this reason, it is important to provide tools to connect these experts with AI technology.

2.3.4 Regions and Availability Zones

An IBM Cloud region is a defined geographical territory that you can deploy your apps to. You can deploy your apps to the region that is nearest to your customers to get low application latency. To address security issues, you can also select the region where you want to keep the application data. When you build apps in multiple regions, if one region becomes unavailable, the apps that are in the other regions continue to run.

Within these regions, there are Availability Zones that are logically isolated data centers with independent power and networks. Most major zones are made up of at least three Availability Zones. Traffic within a zone has ultra-low latency. Figure 2-4 shows the locations of the data centers.

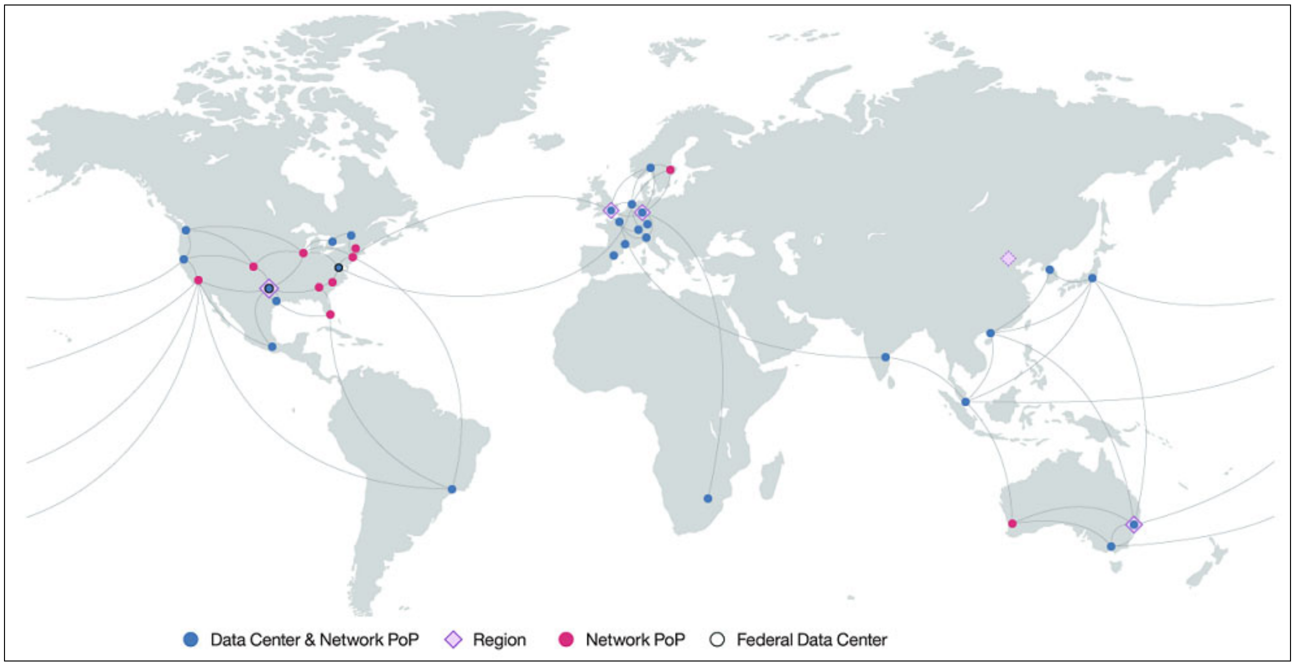


Figure 2-4 Available data centers

2.3.5 Account types

There are three types of IBM Cloud accounts:

- Free
- Pay-Go
- Subscription

When you signed up for your IBM Cloud account, you got a Free account. This type of account is available without a fee and gives you a capped amount of capacity to run apps and services. Many of the services in the catalog have a Lite plan that you can use without a fee for evaluation purposes. Services provisioned under the Lite plan should not be used for development or production scenarios because they do not have sufficient capacity.

You can convert your account to a Pay-Go account by adding a credit card. This account type allows you to create services that cost money. Your credit card will be billed monthly based on your usage.

Note: To leverage the full potential of the sample application that will be presented in this paper, you need an IBM Cloud Pay-As-You-Go account. When creating or upgrading a lite account to a Pay-As-You-Go, IBM Cloud offers a USD 200 credit that can go towards creating, testing, and enhancing this application.

If you do not have an IBM Cloud account, start your free trial to sign up for an IBMid and create your IBM Cloud account. Then, upgrade it to a Pay-As-You-Go Account by adding a credit card to take advantage of the USD 200 credit offer.

Subscription accounts are purchased from IBM by speaking to a sales representative. You commit to a time period and a monthly fee based on estimates, and in return receive a discount. Any credit that is not used in one month rolls over to the next until the end of the subscription term.

2.3.6 Identity and access management

IBM Cloud identity and access management (IAM) enables you to securely authenticate users for platform services and control access to resources consistently across the IBM Cloud platform. A set of IBM Cloud services is enabled to use Cloud IAM for access control. These services are organized into resource groups within your account to give users quick and easy access to more than one resource at a time.

Cloud IAM access policies are used to assign users and service IDs access to the resources within your account. You can group a set of users and service IDs into an access group to easily give all entities within the group the same level of access.

A policy assigns a user or service ID one or more roles with a combination of attributes that define the scope of access. The policy can provide access to a single service down to the instance level, or the policy can apply to a set of resources organized together in a resource group. Depending on the user roles that you assign, the user or service ID will have different levels of access for completing platform management tasks or accessing a service by using the UI or performing specific types of API calls.

Figure 2-5 shows the access control process flow.

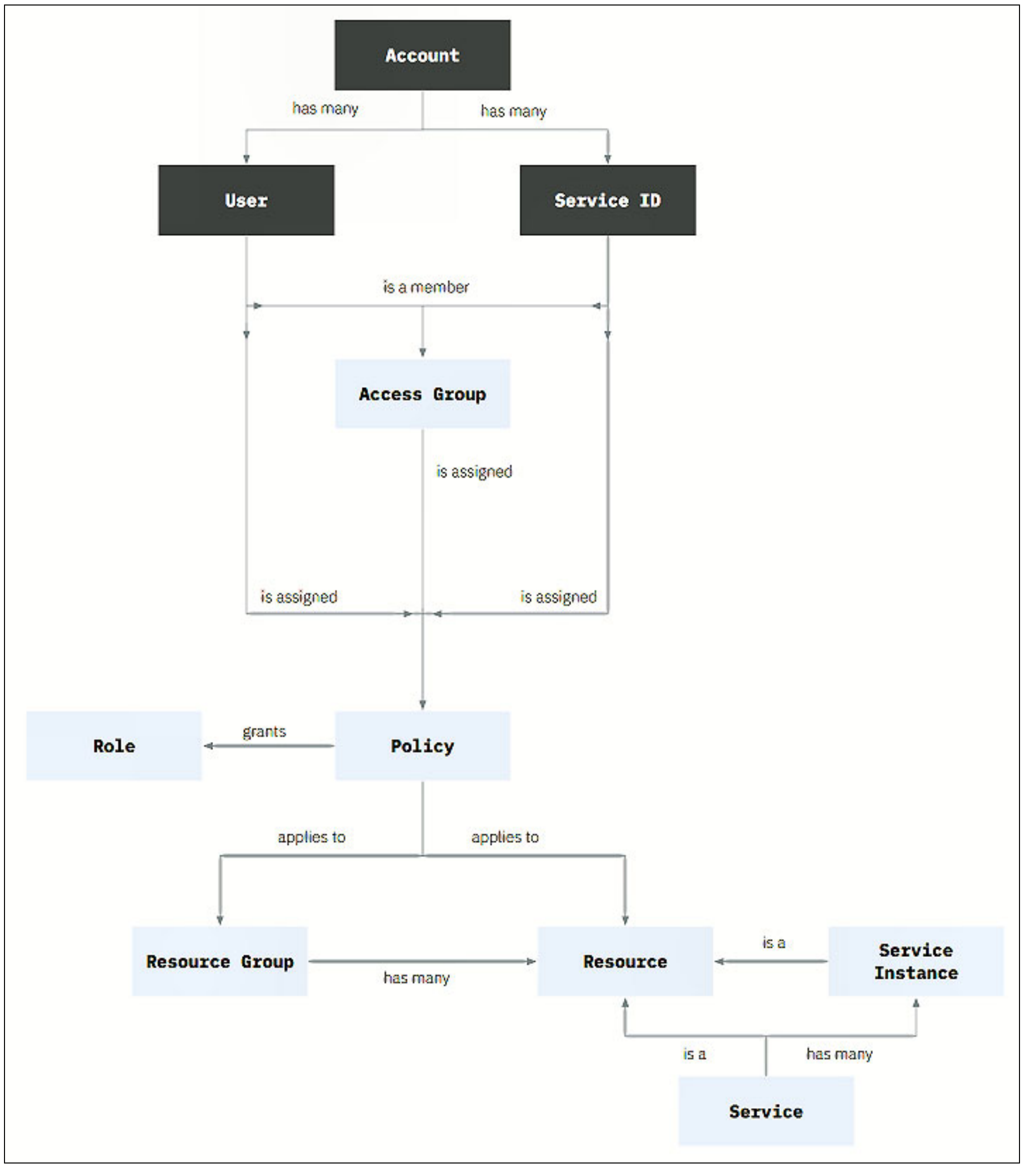


Figure 2-5 Access control diagram

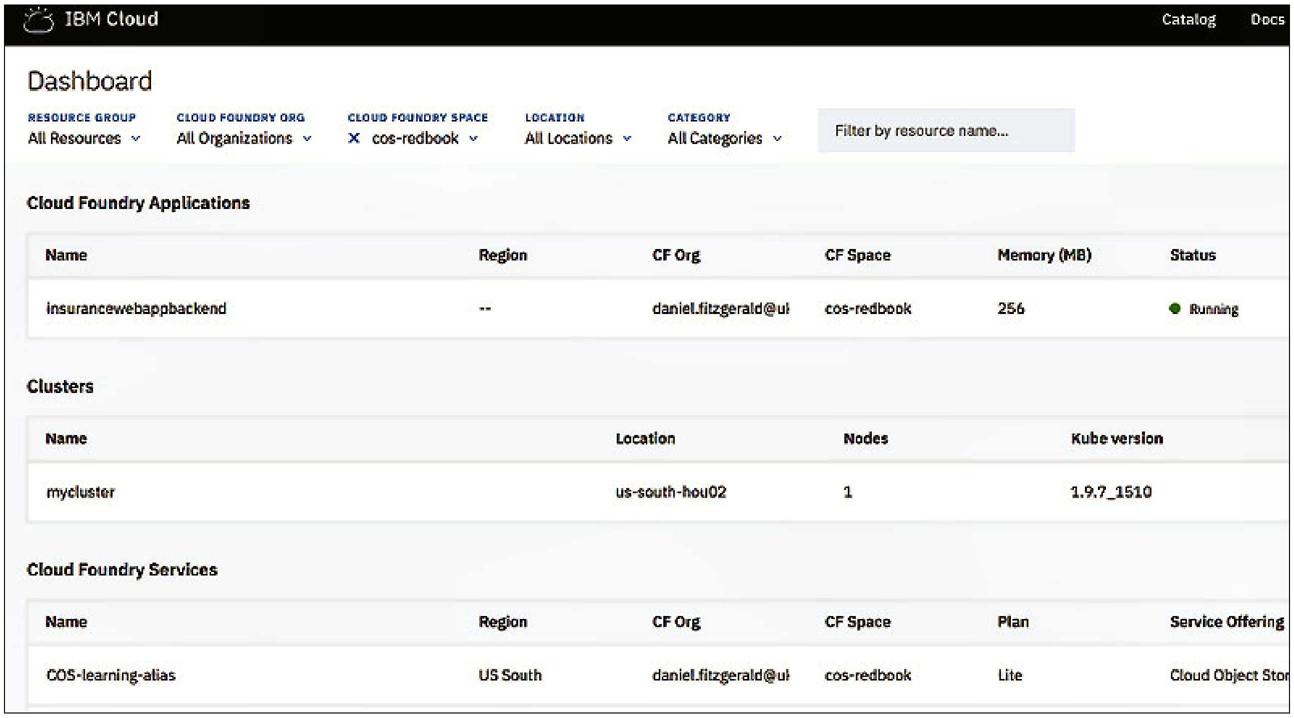
For services that do not support creating Cloud IAM policies for managing access, you can use Cloud Foundry access.

Later in the series, we will be creating IAM policies to give users and apps access to IBM Cloud resources.

2.3.7 Manage IBM Cloud using the console

The primary way that you access IBM Cloud (at least at first) will be using the console. It is available at the IBM Cloud website. The console gives you the following views:

- Overview of your resources (Figure 2-6)


IBM Cloud
Catalog
Docs
Dashboard
RESOURCE GROUP All Resources
CLOUD FOUNDRY ORG All Organizations
CLOUD FOUNDRY SPACE cos-redbook
LOCATION All Locations
CATEGORY All Categories
Filter by resource name...

Cloud Foundry Applications

Name	Region	CF Org	CF Space	Memory (MB)	Status
insurancewebappbackend	--	daniel.fitzgerald@ul	cos-redbook	256	● Running

Clusters

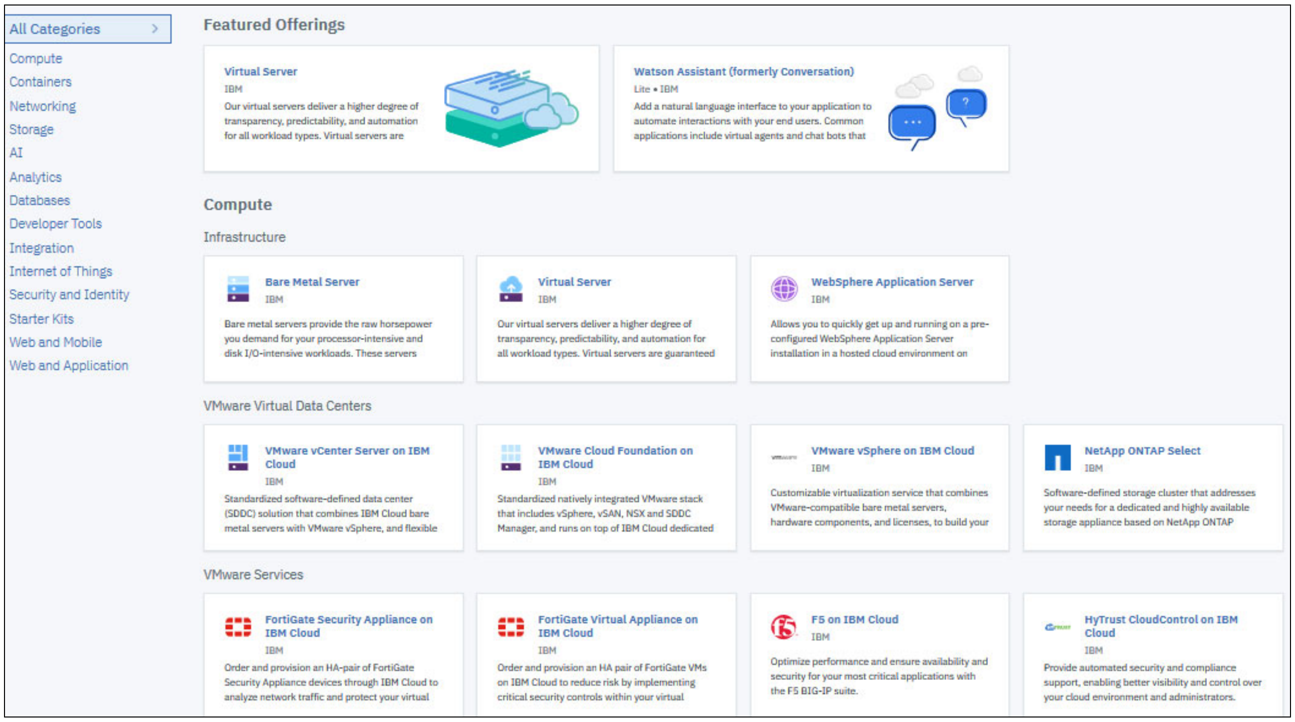
Name	Location	Nodes	Kube version
mycluster	us-south-hou02	1	1.9.7_1510

Cloud Foundry Services

Name	Region	CF Org	CF Space	Plan	Service Offering
COS-learning-alias	US South	daniel.fitzgerald@ul	cos-redbook	Lite	Cloud Object Stor

Figure 2-6 Dashboard to view your IBM Cloud resources

- Catalog window (Figure 2-7)



All Categories

Compute
Containers
Networking
Storage
AI
Analytics
Databases
Developer Tools
Integration
Internet of Things
Security and Identity
Starter Kits
Web and Mobile
Web and Application

Featured Offerings

Virtual Server
IBM
Our virtual servers deliver a higher degree of transparency, predictability, and automation for all workload types. Virtual servers are

Watson Assistant (formerly Conversation)
Lite • IBM
Add a natural language interface to your application to automate interactions with your end users. Common applications include virtual agents and chat bots that

Compute

Infrastructure

Bare Metal Server
IBM
Bare metal servers provide the raw horsepower you demand for your processor-intensive and disk I/O-intensive workloads. These servers

Virtual Server
IBM
Our virtual servers deliver a higher degree of transparency, predictability, and automation for all workload types. Virtual servers are guaranteed

WebSphere Application Server
IBM
Allows you to quickly get up and running on a pre-configured WebSphere Application Server installation in a hosted cloud environment on

VMware Virtual Data Centers

VMware vCenter Server on IBM Cloud
IBM
Standardized software-defined data center (SDDC) solution that combines IBM Cloud bare metal servers with VMware vSphere, and flexible

VMware Cloud Foundation on IBM Cloud
IBM
Standardized natively integrated VMware stack that includes vSphere, vSAN, NSX and SDDC Manager, and runs on top of IBM Cloud dedicated

VMware vSphere on IBM Cloud
IBM
Customizable virtualization service that combines VMware-compatible bare metal servers, hardware components, and licenses, to build your

NetApp ONTAP Select
IBM
Software-defined storage cluster that addresses your needs for a dedicated and highly available storage appliance based on NetApp ONTAP

VMware Services

FortiGate Security Appliance on IBM Cloud
IBM
Order and provision an HA-pair of FortiGate Security Appliance devices through IBM Cloud to analyze network traffic and protect your virtual

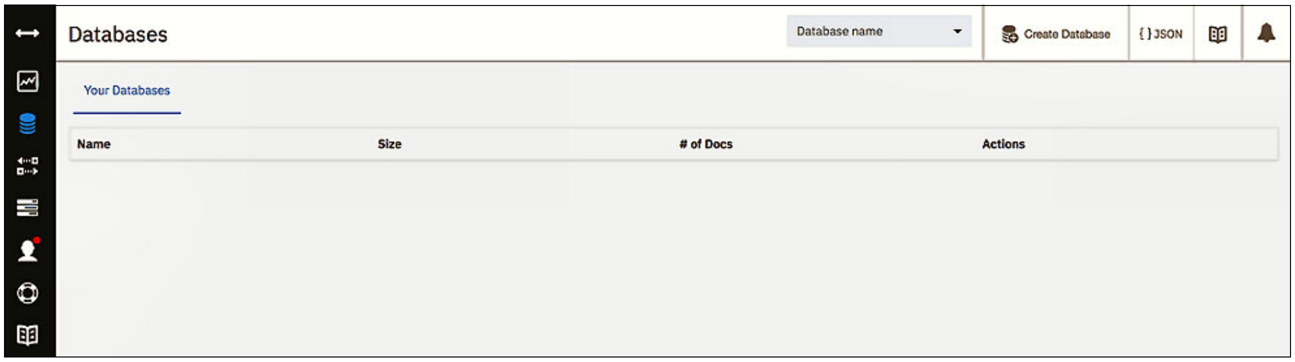
FortiGate Virtual Appliance on IBM Cloud
IBM
Order and provision an HA pair of FortiGate VMs on IBM Cloud to reduce risk by implementing critical security controls within your virtual

F5 on IBM Cloud
IBM
Optimize performance and ensure availability and security for your most critical applications with the F5 BIG-IP suite.

HyTrust CloudControl on IBM Cloud
IBM
Provide automated security and compliance support, enabling better visibility and control over your cloud environment and administrators.

Figure 2-7 Catalog view to create services

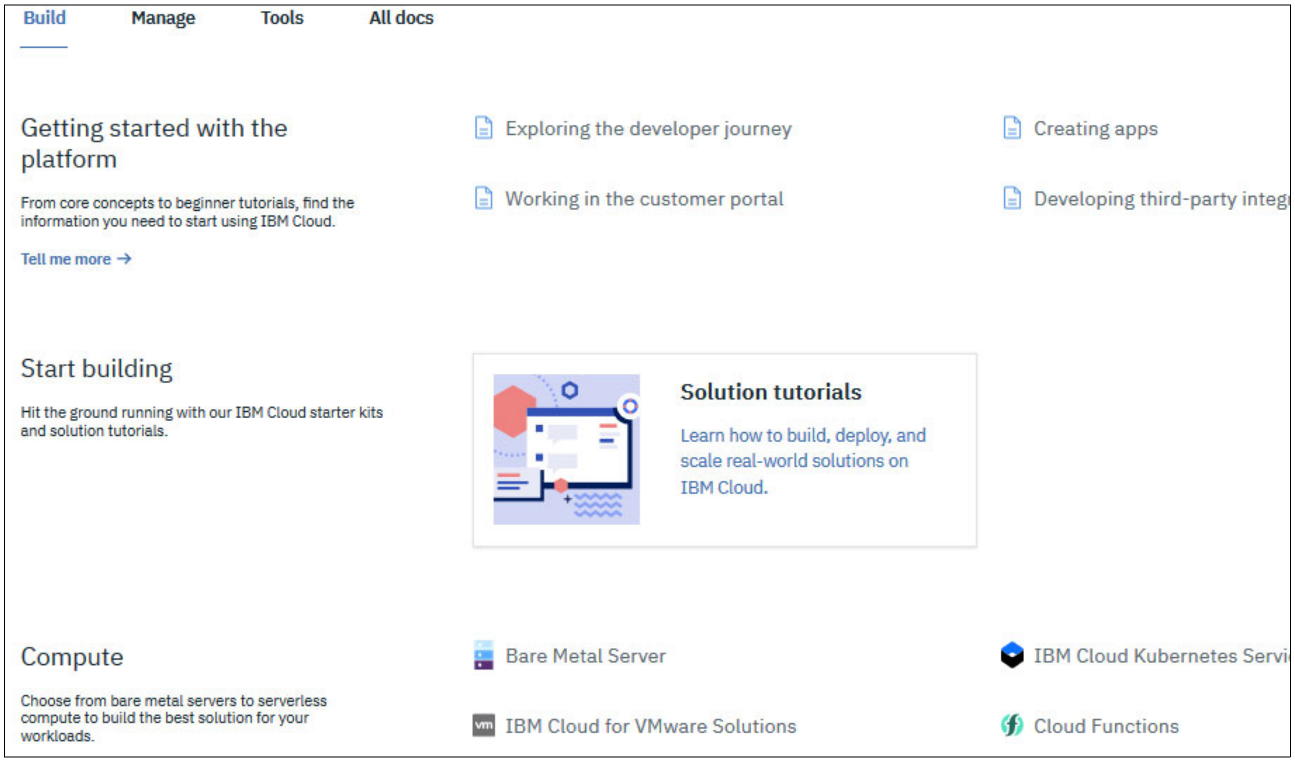
- Configuring and managing services dashboard (Figure 2-8)



Name	Size	# of Docs	Actions

Figure 2-8 Dashboard to configure and manage services

- Documentation (Figure 2-9)


Build
Manage
Tools
All docs

Getting started with the platform

From core concepts to beginner tutorials, find the information you need to start using IBM Cloud.

Tell me more →
Exploring the developer journey
Creating apps
Working in the customer portal
Developing third-party integ

Start building

Hit the ground running with our IBM Cloud starter kits and solution tutorials.

Solution tutorials

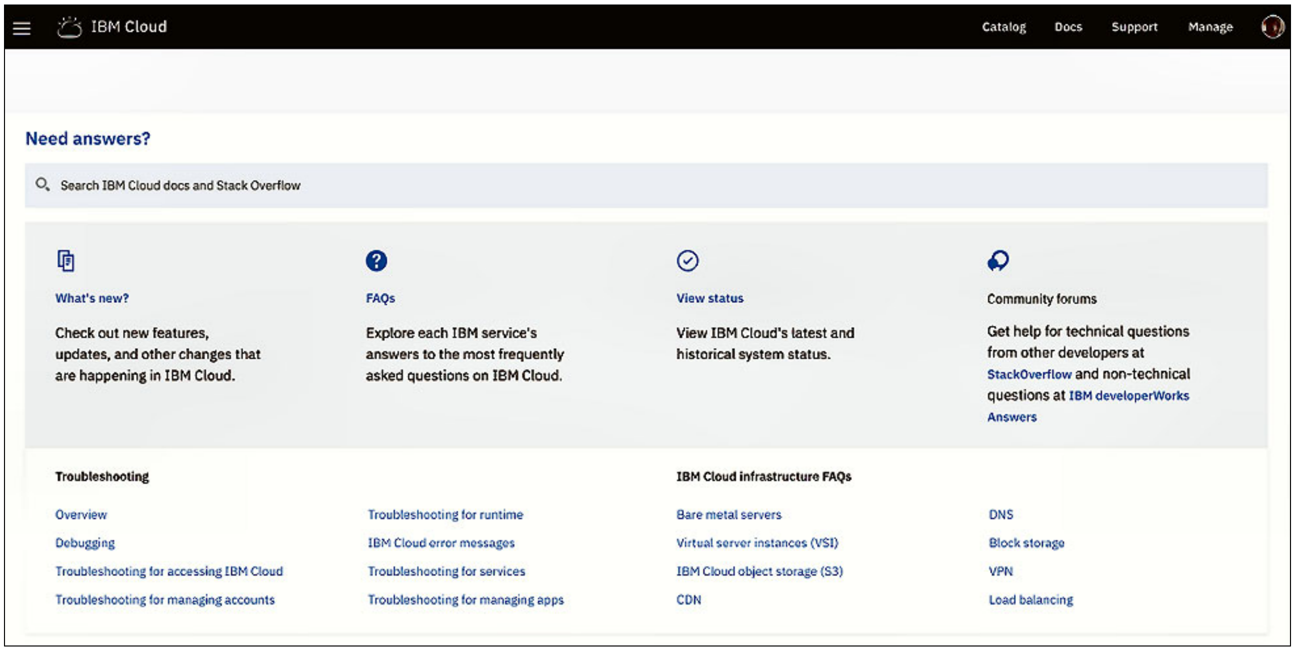
Learn how to build, deploy, and scale real-world solutions on IBM Cloud.

Compute

Choose from bare metal servers to serverless compute to build the best solution for your workloads.

Figure 2-9 Documentation

- Monitoring support tickets (Figure 2-10)



Need answers?

What's new?

Check out new features, updates, and other changes that are happening in IBM Cloud.

FAQs

Explore each IBM service's answers to the most frequently asked questions on IBM Cloud.

View status

View IBM Cloud's latest and historical system status.

Community forums

Get help for technical questions from other developers at StackOverflow and non-technical questions at IBM developerWorks Answers

Troubleshooting

Overview	Troubleshooting for runtime
Debugging	IBM Cloud error messages
Troubleshooting for accessing IBM Cloud	Troubleshooting for services
Troubleshooting for managing accounts	Troubleshooting for managing apps

IBM Cloud infrastructure FAQs

Bare metal servers	DNS
Virtual server instances (VSI)	Block storage
IBM Cloud object storage (S3)	VPN
CDN	Load balancing

Figure 2-10 Support tickets

2.3.8 Managing IBM Cloud using the command line

IBM Cloud offers a command line interface that allows you to manage all aspects of the platform. You should download and install it now. The CLI uses plug-ins that can be added to the CLI to work with different aspects of the platform.

To test the CLI, complete these steps:

1. Log in to the CLI. If you are using a federated ID, you must add the `-sso` flag.

   ```
   $ bx login

   > Choose your API endpoint - 4. us-south - https://api.ng.bluemix.net
   > Enter the email you used to sign up for your IBM ID
   > Enter the password you used when you signed up for your IBM ID
   > Choose your account
   ```

2. Explore all the possible commands that you can run with the CLI:

   ```
   $ bx
   ```

3. Now you need to target an IBM Cloud organization. An organization is a way of logically dividing your IBM Cloud account into distinct areas, each with its own quota and set of users. Target an IBM Cloud organization:

   ```
   $ bx account orgs
   ```

   Select an organization from the list:

   ```
   $ bx target -o <org_name>
   ```

4. List the spaces within your organization. An organization can be further divided into spaces that can contain apps and services. For example, you might divide an organization into spaces for environments such as dev, test, and production. Target a space in the organization:

   ```
   $ bx account spaces
   ```

5. Choose a space:

   ```
   $ bx target -s <space_name>
   ```

6. List the available services in the catalog:

   ```
   $ bx service offerings
   ```

7. Create a Cloudant NoSQL DB instance:

   ```
   $ bx service create cloudantNoSQLDB Lite my-cloudant
   ```

8. Create a service key for Cloudant:

   ```
   $ bx service key-create my-cloudant my-cloudant-keys
   ```

9. Show the Cloudant service keys:

   ```
   $ bx service key-show my-cloudant my-cloudant-keys
   ```

The IBM Cloud CLI uses a plug-in architecture to add additional services. For example, if you want to manage an IBM Containers cluster or deploy a Cloud Functions action, you can install the plug-in by completing these steps:

1. List the available plug-ins:

   ```
   $ bx plugin repo-plugins -r Bluemix
   ```

2. Install the Cloud Functions plug-in from the repo:

   ```
   $ bx plugin install cloud-functions -r Bluemix
   ```

You have now installed the IBM Cloud command line and added a plug-in that allows you to work with Cloud functions. You can install more plug-ins to work with Containers, Cloud Foundry, and so on.

2.4 Best practices and documentation

When building a solution, it is always useful to have guidance and best practices to hand. For more information, see the following resources:

- IBM Cloud Documentation: Documentation for all 180+ services and tutorials, best practices, and tips for setting up your account and administering the platform
- IBM Cloud Architecture center: Best practices, code samples, videos, and so on from experts across IBM, captured from years of building and operating cloud solutions
- IBM Cloud solution tutorials: A range of end-to-end tutorials on everything from deploying a LAMP stack to analyzing and visualizing data using Apache Spark

2.5 Next steps

The next module introduces the application that you will build over the course of the series.

2.6 Other references

Here are some useful links for further reading:

- IBM Cloud docs

  `https://console.bluemix.net/docs/`
- IBM Cloud Architecture Center

  `https://www.ibm.com/cloud/garage/architectures/`
- IBM Cloud Garage Method

  `https://www.ibm.com/cloud/garage/#changeHowYouWorkSection`
- IBM Cloud Solution Tutorials

  `https://console.bluemix.net/docs/tutorials/index.html#tutorials`

3

Application design and architecture

Welcome to part three in this tutorial series! This series explains how to build a cloud-native, consumer-facing Insurance application on IBM Cloud that is built according to best practices. It uses IBM Cloud Object Storage at its core, and employs AI and machine learning to provide a unique digital experience for a customer.

We have chosen this example because most people are familiar with filing claims after an automobile accident and can easily relate to the steps involved. Another important point is that the scenario is broadly applicable to many other industries and areas. Therefore, it is a good example of how to apply IBM Cloud Object Storage to case management solutions that involve large data assets can be cost effectively stored in IBM Cloud Object Storage.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

3.1 Learning objectives

After completing this module, you will be able to:

- Describe the architecture of the claims application.
- Explain the scenario implemented and how the application adds value.
- Describe the different IBM Cloud services used in the architecture of this application.
- Identify how each service adds value for the particular use case.

3.2 Getting started

Before beginning this module, be sure to complete Chapter 2, “IBM Cloud primer” on page 11.

3.3 Implementation

At the beginning of the claim process, an automobile accident occurs and the insured person submits a claim for compensation. Along with submitting their loss notice, the customer typically includes additional documentation such as pictures of the damage and sketches of the road layout. In the past, this claim would be done by filling out a paper document. In today's world, the notice can occur through a phone call, an online form, a mobile app, or several other methods.

Automobile accidents can range from a single car fender-bender to a serious incident involving loss of life and criminal charges. They can also involve financial fraud if they include false information.

Although the notice can be received through various electronic channels, many steps in the processing of claims are still manual.

The goal of Insurance Company X's claims application is to automate many of these steps with AI to assist both customers during claim creation and caseworkers while they are resolving the cases. Insurance X also wants to create a scalable and cost-effective solution that can serve as a base for future innovation.

Each claim is different, even though the incidents and characteristics of the claims might be similar. Which activities are needed for each claim depends on the details of the claim.

For a customer, the beginning of this process will always be very similar. The first step is always to open a case and provide some initial information. This information will typically include providing pictures of the insured vehicle along with a description of the damage and what happened. Assisting customers with analyzing image content and adding the location and weather conditions speeds up the creation of a claim, reduces friction, and improves customer satisfaction.

For a caseworker processing the claim after it has been submitted, the following information can be provided to assist the resolution of the claim:

- Are all necessary pictures of the insured car provided (front/left/right/back)?
- At which location did the accident happen?
- What was the weather like when the accident happened?
- What is the estimated cost for repair?

In addition, the application can detect suspicious items in the data provided to assist with fraud detection:

- Have the pictures been taken at the place that the accident happened?
- Are all pictures showing the actual car of the insured person?
- Is the weather described consistent with the weather recorded at that time (such as for a hailstorm damage claim)?
- Do the pictures match other pictures of the same car from previous claims?

You can identify other areas where AI services can provide value and speed up the process, lower error rates, and support fraud detection.

Scenario:

Alice is driving on a roadway in a snow storm when her car strikes a large object. Although she is not hurt, her car is too damaged to drive. The police arrive, and a tow truck is dispatched. When Alice files the claim with the Insurance X web application, she is prompted to provide pictures of her car from all four angles and a short description of what happened. When she adds the pictures, the angle is automatically detected along with other details about the location and weather, and the car is matched against the cars assigned to her account so that she can quickly complete the claim.

The claim is assigned to Insurance X employee Bob, who checks the claim details populated by the system. Because all of the details provided by Alice match the information extracted, the system sets the flag for possible fraud to *false* and suggests that Bob approve proceeding with getting an estimate for the repair.

Solution:

The architecture of the solution that enables these scenarios is guided by both the functional requirements and non-functional requirements such as deployment into multiple regions around the globe and down-time-free deployments, as shown in Figure 3-1.

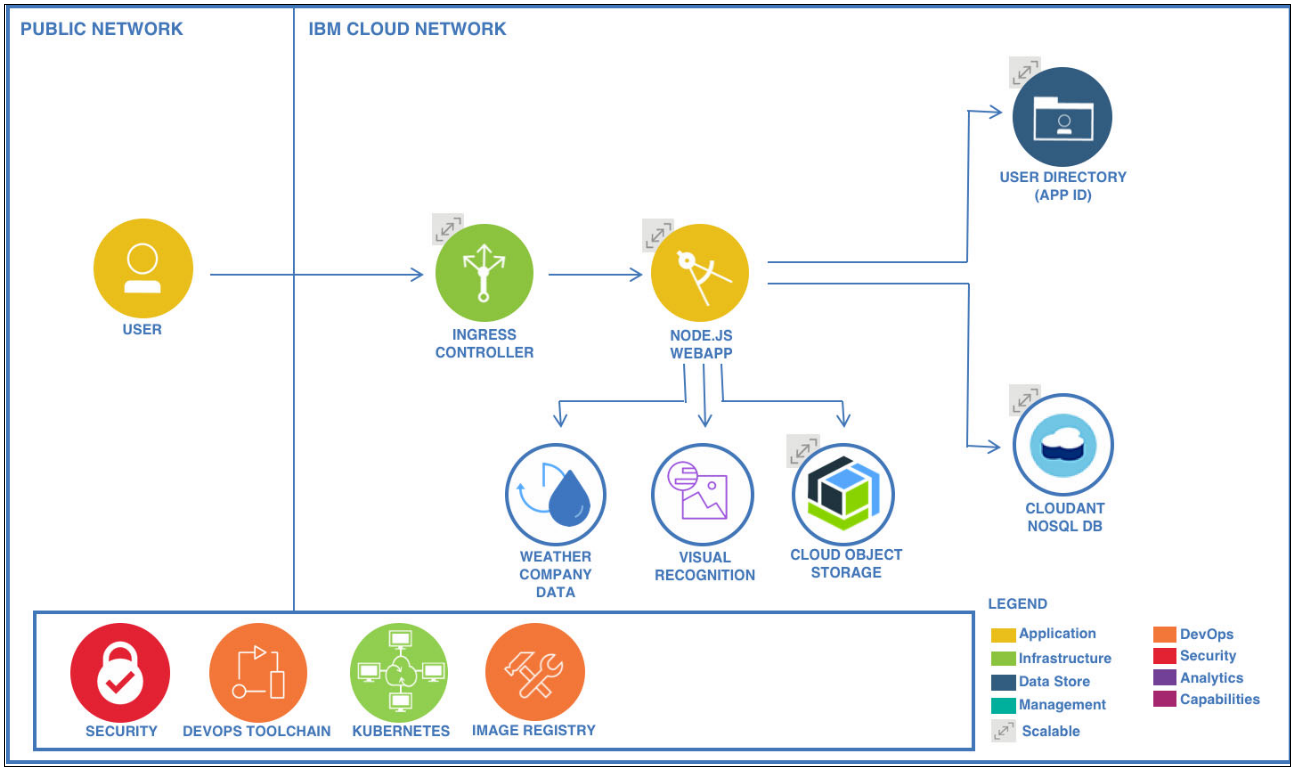


Figure 3-1 Solution architecture

At the core, the application consists of a Node.js based UI and API microservice that run in Docker containers deployed in one or more IBM Container Service clusters.

We use Node.js because it allows code to be shared between the front end and back end, as well as easily enabling advanced techniques such as server-side rendering. Because we are focusing on integrating with IBM Cloud Object Storage and other services in this tutorial, we are not using some of the more advanced Node.js patterns and technologies like Webpack, React, Babel, Redux, Responsive Design, and Backend for Frontend that you would see in a production service.

To keep things simple, we use Bootstrap.js to build the front end as single-page application (SPA) that serves users. This application allows users to add new claims and list existing claims.

Images that are uploaded as part of a claim are normalized to a fixed set of dimensions and then stored in both original and normalized variants in IBM Cloud Object Storage cross-regional buckets to provide cost-effective, resilient, long-term storage of data. We use the normalized version of images for all of the processing, while still keeping the original image for reference and audit purposes.

Claim related metadata is stored in IBM Cloudant, which is a schema free JSON document store. By using Cloudant and continuous replication, you can spread the application throughout multiple geographically dispersed regions, while still running in an active-active mode. Because every node in a Cloudant cluster can write, the global state of a database is eventually consistent, but not immediately.

The schema free nature of Cloudant allows you to easily and quickly enhance the claim data model to quickly increase the value delivered by the claims application.

An example of such integration is metadata extracted from images or the integration of other Watson services, such as visual recognition, machine learning, and weather data.

Claim metadata for finalized claims is archived to IBM Cloud Object Storage for long-term storage and analytics. By using IBM SQL query, data scientists can use ANSI SQL to gain insight from previous claims. Furthermore, we are enabling the use of other Watson Studio services like Notebooks.

Authentication of users is done using IBM App ID. Besides providing a directory integration, App ID enables the application to support social login by using Facebook or Google with ease.

We use DevOps toolchain to provide integrated, open, and easily extensible continuous integration pipeline. Docker images built by the pipeline will be stored in image registry, where they are scanned for vulnerable packages to prevent known attack vectors and increase security of the application.

The remaining modules in this series incrementally build this application to implement the data flow for claims, as shown in Figure 3-2.

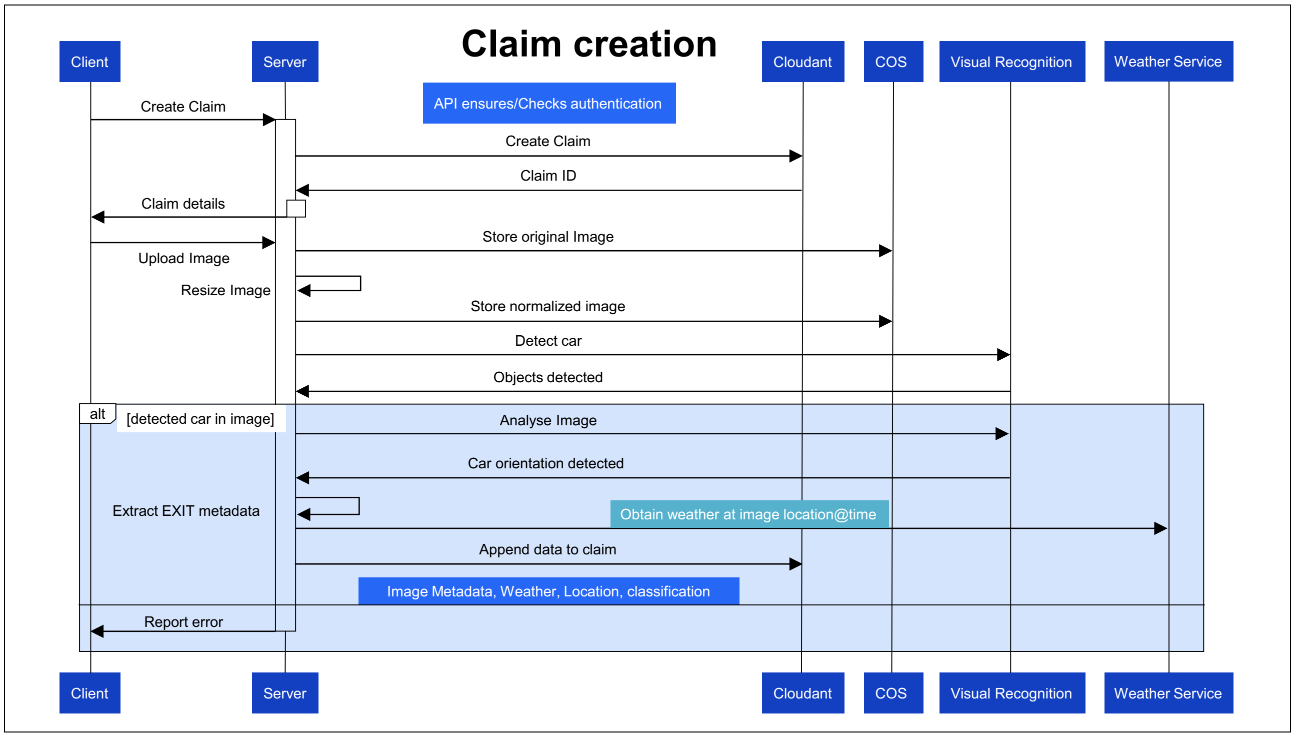


Figure 3-2 Data flow for claim creation

3.4 Next steps

The next module in this series walks you through creating an initial skeleton application using IBM Starter Kits. You will also create an initial user interface.

3.5 Other references

- IBM Cloud Tutorials

  `https://console.bluemix.net/docs/tutorials/index.html#tutorials`

4

Scaffolding using a Starter Kit

Welcome to part four in this tutorial series! This series explains how to build a cloud-native, consumer-facing car insurance claim application on IBM Cloud that is built according to best practices. It uses IBM Cloud Object Storage at its core, and employs AI and machine learning to provide a unique digital experience for a customer.

This module covers Starter Kits. Many options are available to the reader to build a web application using the IBM Cloud ecosystem. A Starter Kit is the lowest barrier entry point to an application developer. The Starter Kit configurator can set up your Cloud native app and DevOps environment in minutes.

Building a web application with all the supporting DevOps infrastructure on the cloud can be intimidating. To make application creation easier, IBM Cloud provides IBM Cloud App Service to help you while building cloud native apps. The IBM Cloud App Service provides starter kits that walk you through the process of creating a “Hello World” Application with all of the required resources.

IBM Cloud offers various Starter Kits that cover various languages, frameworks, and tools that can be applied to insurance application web application.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

4.1 Learning objectives

At the end of this chapter, you will be able to jumpstart a web application with a Starter Kit and add the resources on the IBM Cloud for the application and deploy it on a cluster.

4.2 Getting started

Before you begin, sign up for a free IBM Cloud account or if you already have one, sign in. Note that to leverage the full potential of the sample application that will be presented in this Tutorial series, you need an IBM Cloud Pay-As-You-Go account. When creating or upgrading a lite account to a Pay-As-You-Go, IBM Cloud offers a USD 200 credit that can go towards creating, testing, and enhancing this application.

To leverage the full potential of the sample application that will be presented in this Tutorial series, you need an IBM Cloud Pay-As-You-Go account. When creating or upgrading a lite account to a Pay-As-You-Go, IBM Cloud offers a USD 200 credit that can go towards creating, testing, and enhancing this application.

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module04
```

4.3 Implementation

The following sections cover going from an intention to build an application to an actual working application with all the infrastructure in place, so you can focus on development and not spend time worrying about the infrastructure.

4.3.1 Choosing your Deployment

IBM Cloud offers Cloud Foundry and Kubernetes PaaS for your web application.

Cloud Foundry allows users to deploy and scale application without manual configuration. For more information about IBM Cloud Foundry offerings, see the Cloud Foundry website.

IBM Cloud also provides Kubernetes PaaS for an intuitive user experience and built-in security for rapid delivery of applications. You can bind these applications to cloud services related to Watson, IoT, DevOps, and Data Analytics. As a certified K8s provider, IBM Cloud Container Service provides intelligent scheduling, self-healing, horizontal scaling, service discovery and load balancing, automated rollouts and rollbacks, and secret and configuration management. This Kubernetes service also has advanced capabilities that provide simplified cluster management, container security and isolation policies, the ability to design your own cluster, and integrated operational tools for consistency in deployment. To learn about the IBM Container Service offering, see the Container Service website.

Using Cloud Object Storage with Kubernetes: The default storage on Kubernetes nodes is ephemeral. This type means data is lost when the nodes are rebooted. Using Cloud Object Storage provides persistent storage to the applications that run on Kubernetes nodes.

Why Kubernetes

Kubernetes provides the following advantages:

- Better platform utilization, which gives greater control of the platform to developers.
- Kubernetes is open source with wide vendor support.
- Extensible framework. which allows you to add your own extensible components.

4.3.2 Starter Kits

To build Insurance Company X's Web Application, we had several choices. We decided to use Express.js framework on Node.js as a starting point in this exercise. Express.js has become the de facto standard server framework for Node.js.

At this point, two Starter Kits are available: Express.js Basic and Express.js Basic with React. We chose Express.js Basic because it provides the building block on which we can add components to build a full-fledged application.

Choose the Starter Kit

To select the Starter Kit, complete these steps:

1. Log in to the IBM Cloud and select **Web Apps** on the console, as shown in Figure 4-1.

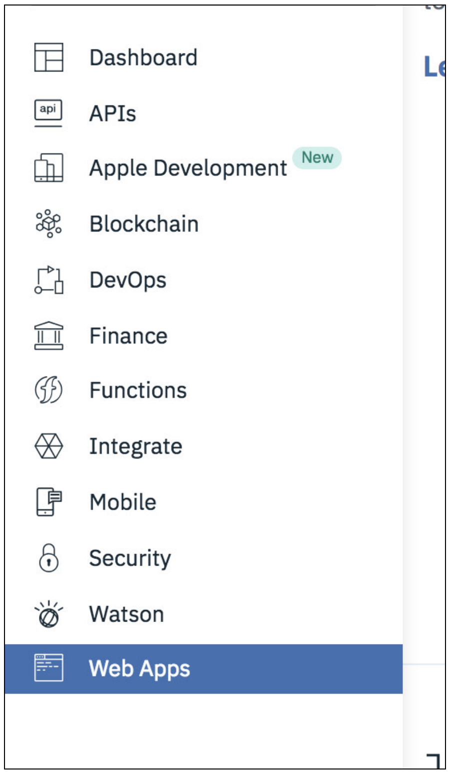
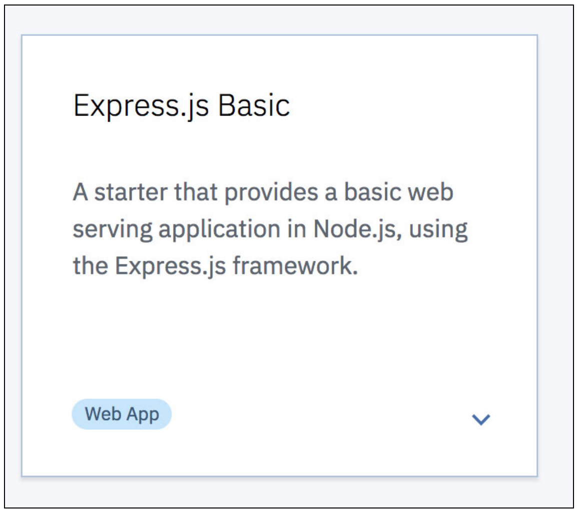


Figure 4-1 Selecting Web Apps

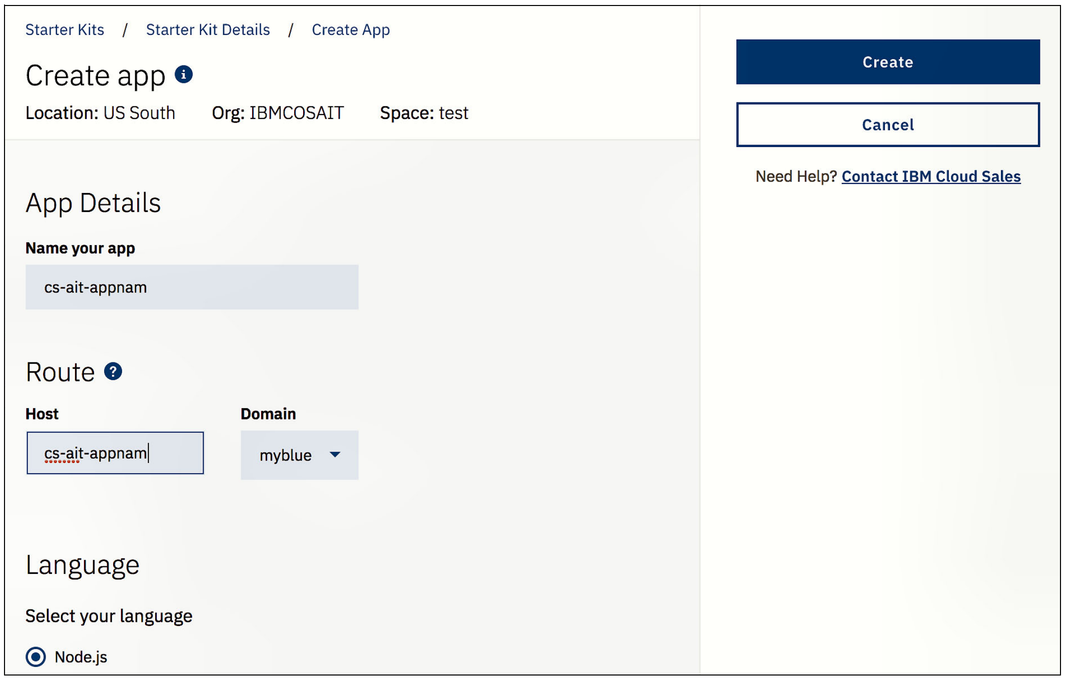
2. Select the **Express.js Basic** Starter Kit, as shown in Figure 4-2.


Express.js Basic

A starter that provides a basic web serving application in Node.js, using the Express.js framework.

Figure 4-2 Selecting the Express.js Basic kit

3. Enter the name of the application and the route to your application, as shown in Figure 4-3.


Starter Kits / Starter Kit Details / Create App
Create app
Create
Cancel
Need Help? Contact IBM Cloud Sales

Location: US South Org: IBMCOSAIT Space: test

App Details

Name your app: cs-ait-appnam

Route

Host: cs-ait-appnam

Domain: myblue

Language

Select your language

Node.js

Figure 4-3 Entering the application name and route

4. Add the App ID, which provides Federated Identity Management, to the application by clicking **Add Resource** → **Security** → **App ID**, as shown in Figure 4-4.

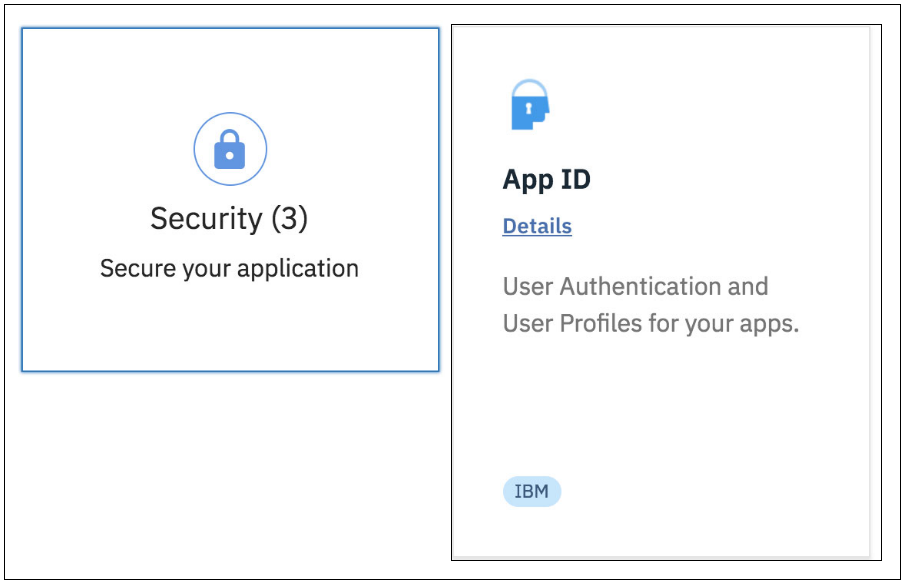


Figure 4-4 Adding the App ID

5. Add Cloudant Database, which provides a repository for metadata storage, to the application by clicking **Add Resource** → **Data** → **Cloudant Database**, as shown in Figure 4-5.

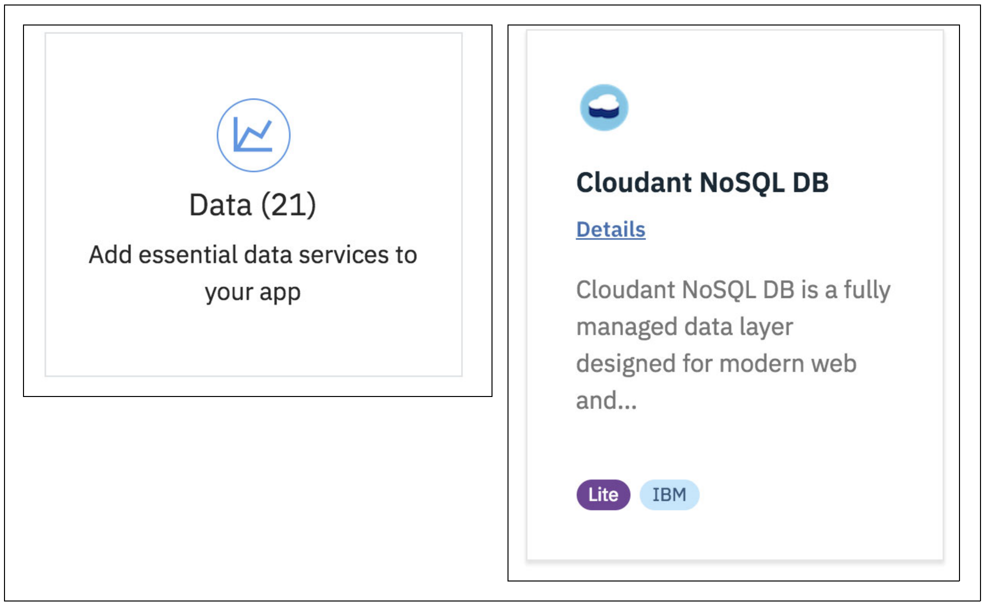

Data (21)
Add essential data services to your app
Cloudant NoSQL DB
Details
Cloudant NoSQL DB is a fully managed data layer designed for modern web and...
Lite
IBM

Figure 4-5 Adding the Cloudant Database

6. Add Object Storage, which provides storage for images and metadata backups, to the application by clicking **Add Resource** → **Storage** → **Cloud Object Storage**, as shown in Figure 4-6.

Note: If you already have a Cloud Object Storage that you intend to use, follow the steps documented in 4.3.6, “Adding an IBM Cloud Object Storage service after deployment” on page 44.

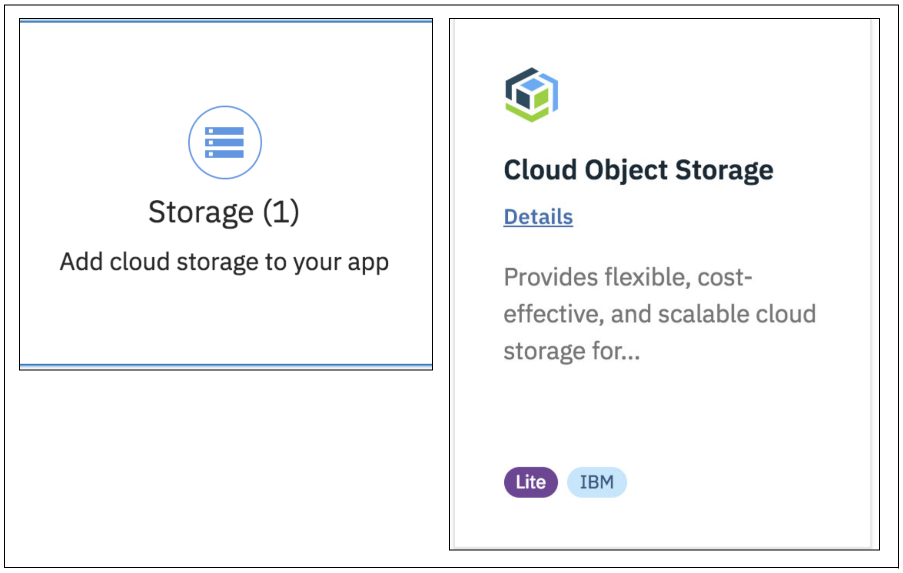

Storage (1)
Add cloud storage to your app
Cloud Object Storage
Details
Provides flexible, cost-effective, and scalable cloud storage for...
Lite
IBM

Figure 4-6 Adding Object Storage

7. Add Watson Visual Recognition, which provides the ability to analyze images, to the application by clicking **Add Resource** → **Watson** → **Visual Recognition**, as shown in Figure 4-7.

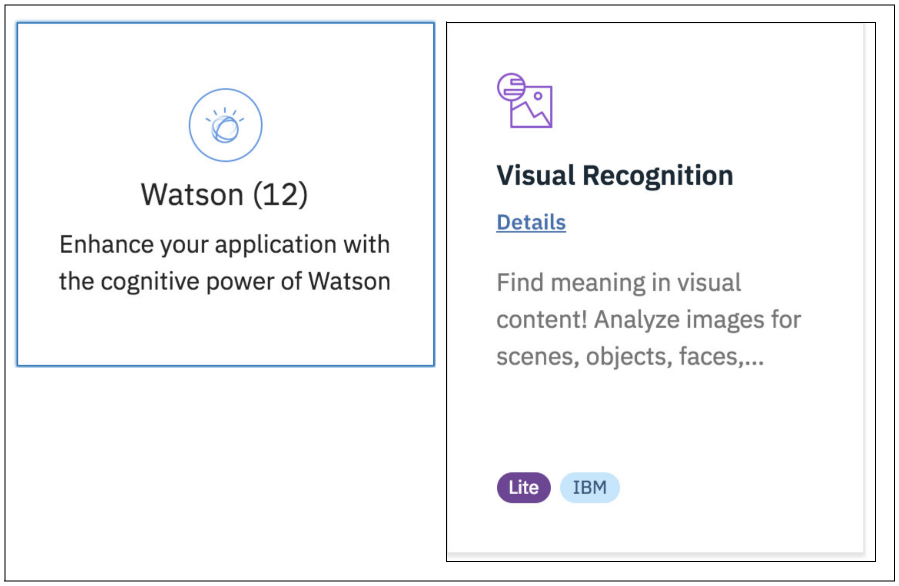

Watson (12)
Enhance your application with the cognitive power of Watson
Visual Recognition
Details
Find meaning in visual content! Analyze images for scenes, objects, faces,...
Lite
IBM

Figure 4-7 Adding Watson Visual Recognition

8. Add Watson Knowledge Studio, which provides Machine Learning, to the application by clicking **Add Resource** → **Watson** → **Knowledge Studio**, as shown in Figure 4-8.

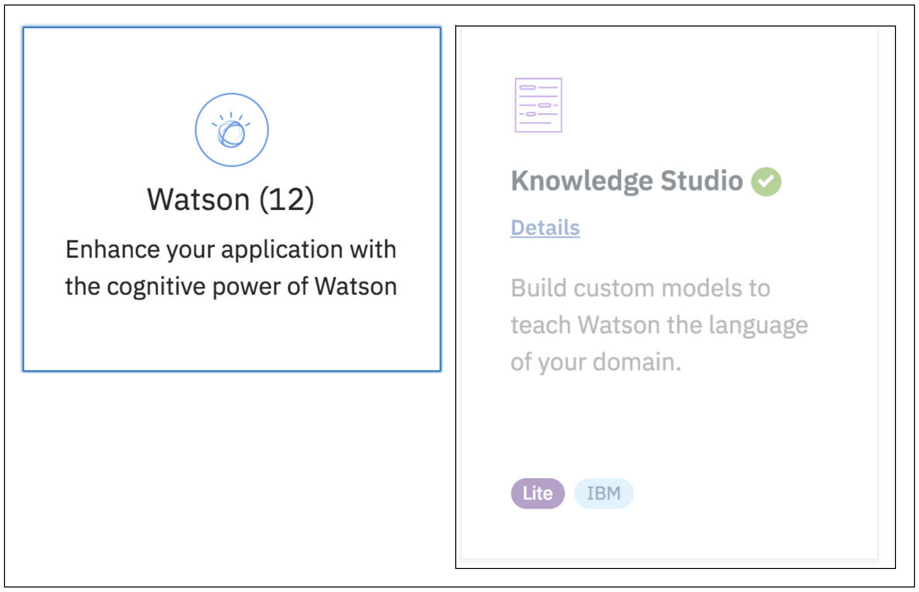

Watson (12)
Enhance your application with the cognitive power of Watson
Knowledge Studio
Details
Build custom models to teach Watson the language of your domain.
Lite
IBM

Figure 4-8 Adding Watson Knowledge Studio

9. Add Weather Data, which provides weather information based on location, to the application by clicking **Add Resource** → **Data** → **Weather Company Data**, as shown in Figure 4-9.

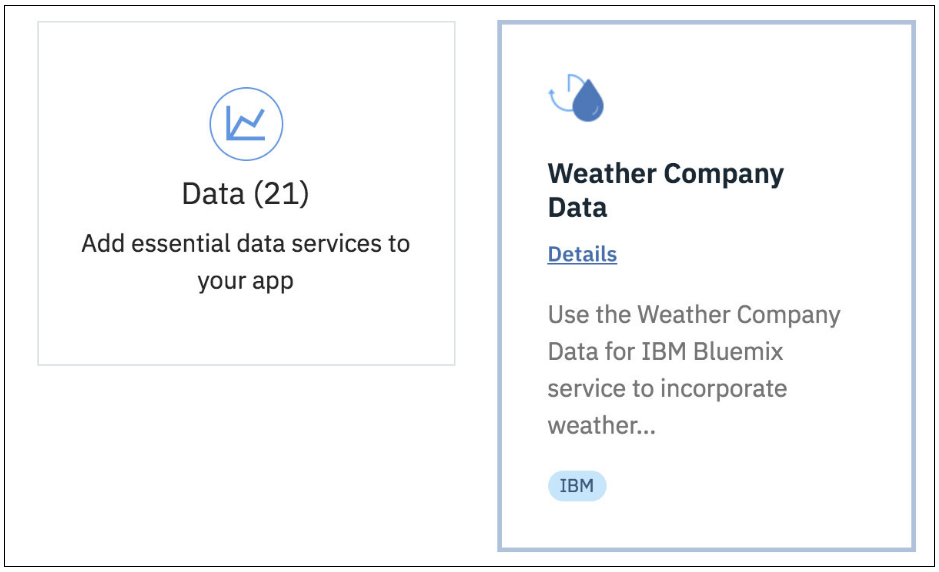
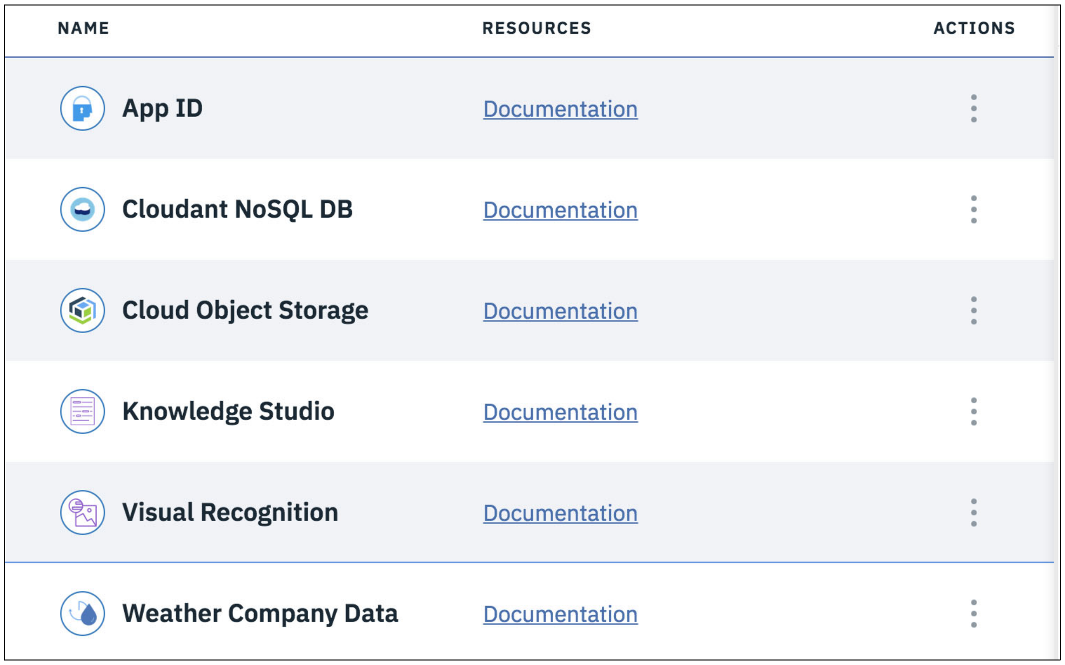

Data (21)
Add essential data services to your app
Weather Company Data
Details
Use the Weather Company Data for IBM Bluemix service to incorporate weather...
IBM

Figure 4-9 Adding Weather Company Data

After adding the services, your resources will look as shown in Figure 4-10.



NAME	RESOURCES	ACTIONS
App ID	Documentation	⋮
Cloudant NoSQL DB	Documentation	⋮
Cloud Object Storage	Documentation	⋮
Knowledge Studio	Documentation	⋮
Visual Recognition	Documentation	⋮
Weather Company Data	Documentation	⋮

Figure 4-10 Completed resources

10. Deploy the services on your cluster, as shown in Figure 4-11. Generally, create the cluster before you create the application on the cluster.

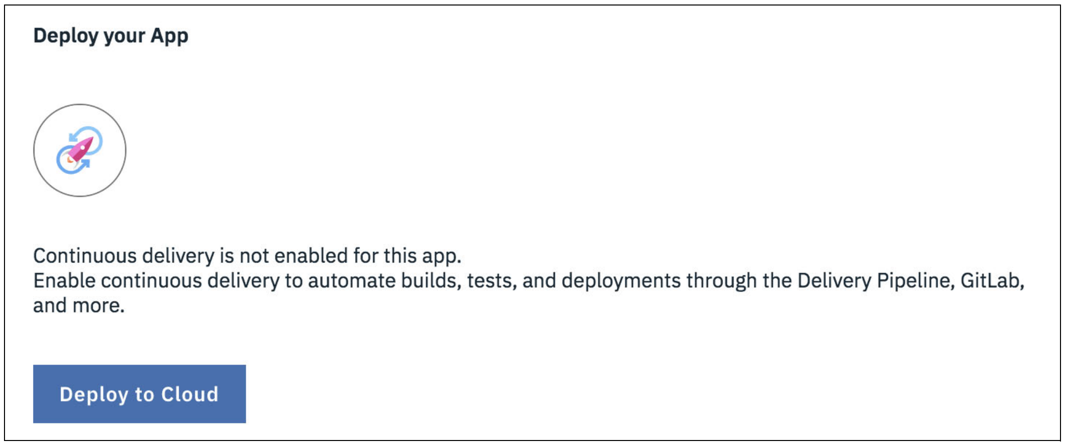
Note: IBM Cloud container registry allows 500 MB of transfer per month. If you encounter a quota error, you need to upgrade your container registry plan.

You can try to delete some images in a repository by running the **`bx cr images`** command to get the namespace of the repository, and then running this command:

```
bx rm namespace-rm <namespace>
```

You can check your quota by running this command:

```
bx cr quota
```



Deploy your App

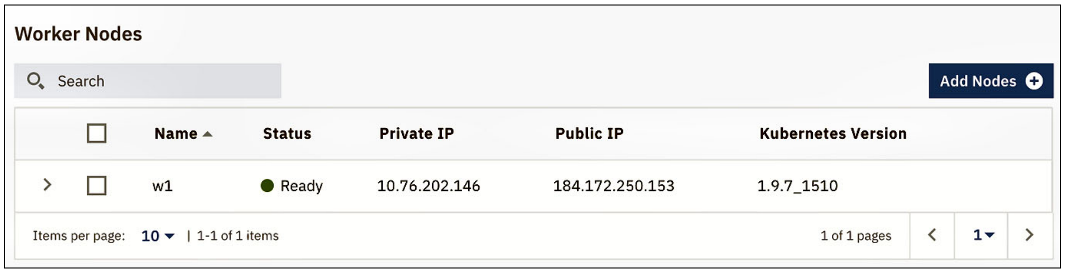
Continuous delivery is not enabled for this app.
Enable continuous delivery to automate builds, tests, and deployments through the Delivery Pipeline, GitLab, and more.

Figure 4-11 Deploying services on your cluster

Note: Free clusters are not available in all regions. Ensure that you are in the region that allows creating free clusters. See the documentation for "Getting started with IBM Cloud Kubernetes Service" on your IBM Cloud console.

After deployment of the application, you will be able to accomplish these tasks:

- Get the public IP of the cluster by navigating to the Worker Nodes window by clicking **Cluster** → **Worker Nodes**, as shown in Figure 4-12.

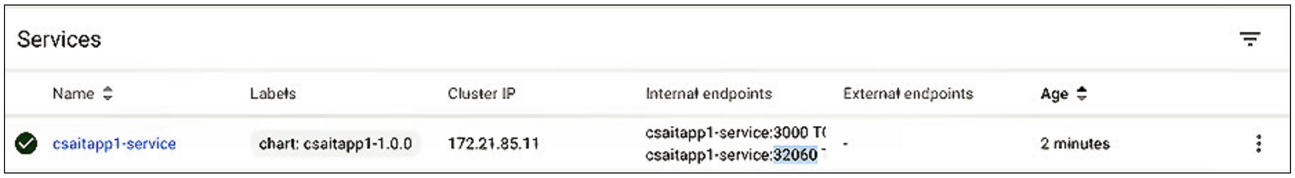


Worker Nodes

	Name	Status	Private IP	Public IP	Kubernetes Version
>	w1	● Ready	10.76.202.146	184.172.250.153	1.9.7_1510

Figure 4-12 Retrieving the public IP

- Obtain a NodePort, which will be the port on which the application will be exposed to on the Worker Node, as shown in Figure 4-13. The Node Port can be obtained by navigating to the Services section of your Kubernetes dashboard.

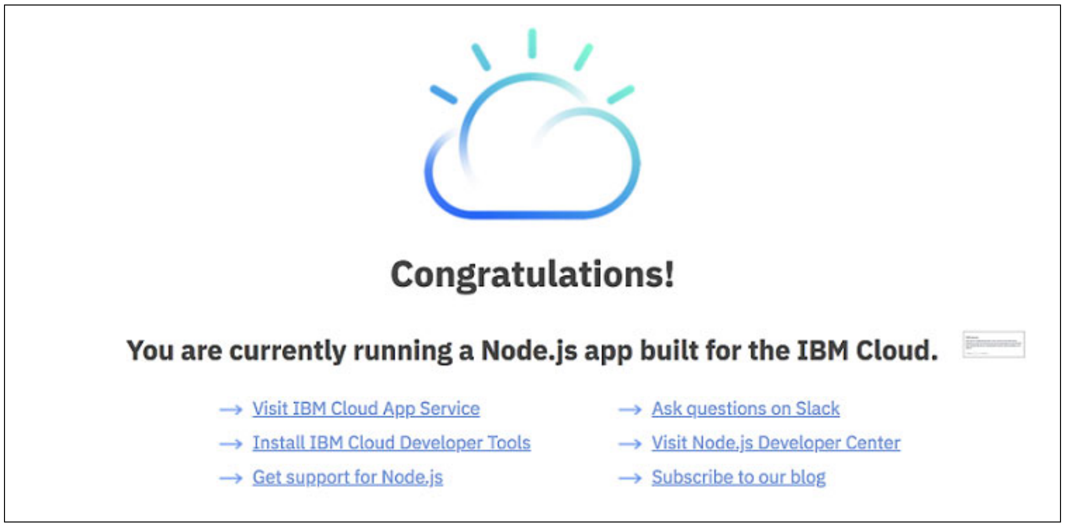


Services

Name	Labels	Cluster IP	Internal endpoints	External endpoints	Age
csaitapp1-service	chart: csaitapp1-1.0.0	172.21.85.11	csaitapp1-service:3000 T(csaitapp1-service:32060	-	2 minutes

Figure 4-13 Retrieving the NodePort

- Start the web application by pointing your browser to `http://<Public IP of your cluster>:<NodePort>`, as shown in Figure 4-14.



Congratulations!

You are currently running a Node.js app built for the IBM Cloud.

→ Visit IBM Cloud App Service
→ Install IBM Cloud Developer Tools
→ Get support for Node.js
→ Ask questions on Slack
→ Visit Node.js Developer Center
→ Subscribe to our blog

Figure 4-14 Starting the web application

- View Development Tool Chain on the console with the necessary DevOps tools. IBM Cloud provides ways to add your own tools to this tool chain, as shown in Figure 4-15.

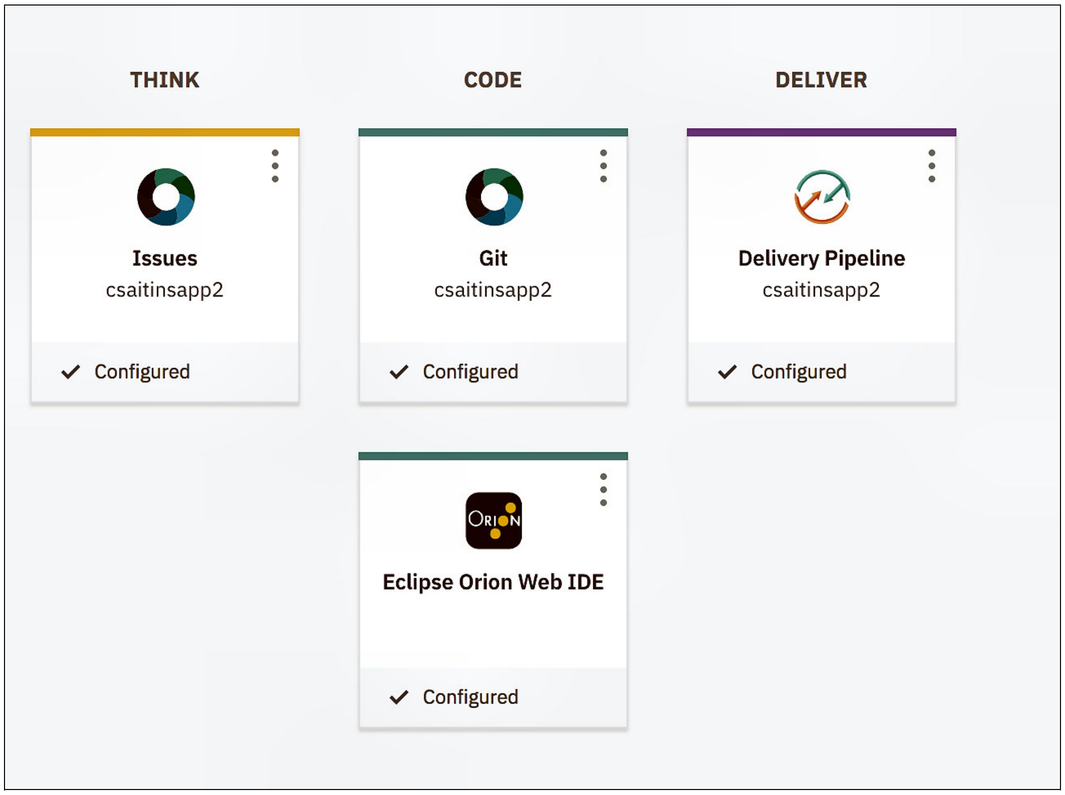
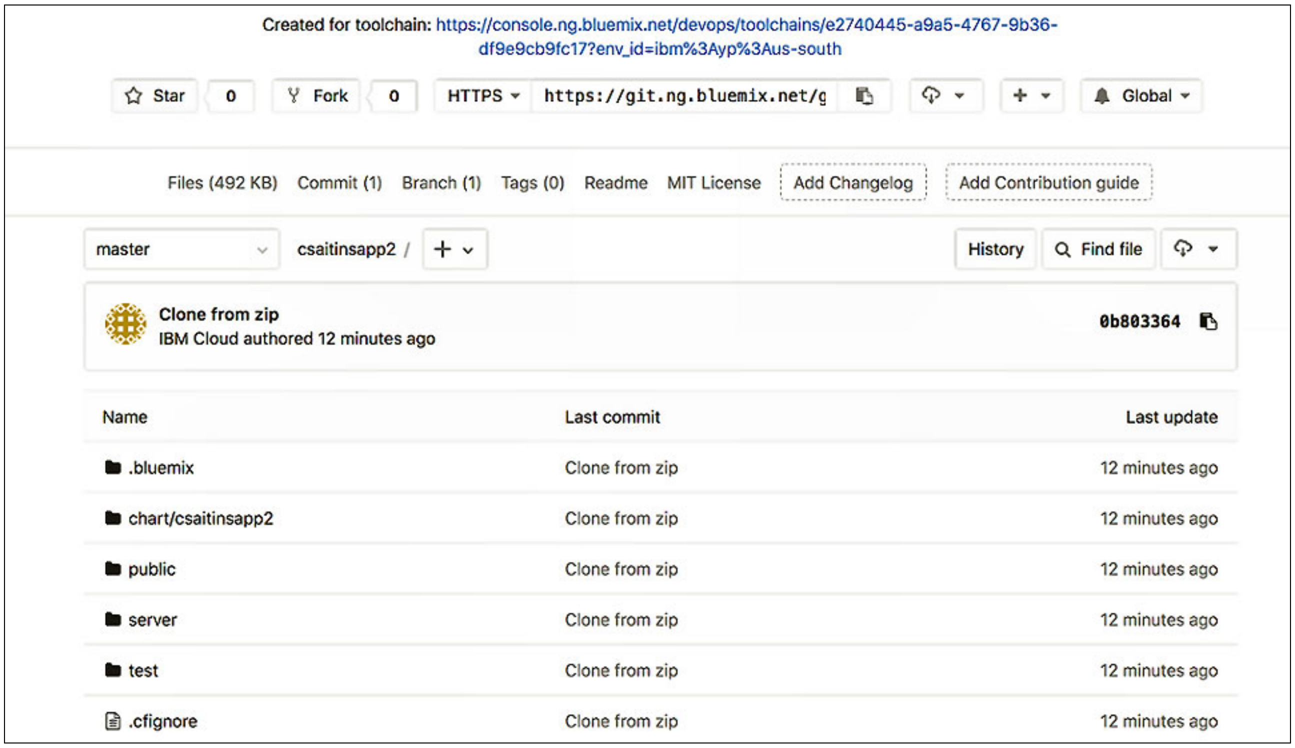


Figure 4-15 Viewing the Development Tool Chain

- Check out the code and perform development activities by navigating to the Git tile in the tool chain and running Git commands. It also provides the Eclipse Orion Web IDE for Development in the web browser, as shown in Figure 4-16.



Created for toolchain: https://console.ng.bluemix.net/devops/toolchains/e2740445-a9a5-4767-9b36-df9e9cb9fc17?env_id=ibm%3Ayp%3Aus-south

Clone from zip
IBM Cloud authored 12 minutes ago
0b803364

Name	Last commit	Last update
.bluemix	Clone from zip	12 minutes ago
chart/csaitinsapp2	Clone from zip	12 minutes ago
public	Clone from zip	12 minutes ago
server	Clone from zip	12 minutes ago
test	Clone from zip	12 minutes ago
.cfignore	Clone from zip	12 minutes ago

Figure 4-16 Eclipse Orion Web IDE

- Customize the delivery pipeline

4.3.3 Delivery pipeline

A delivery pipeline automates continuous development of a project. In a project's pipeline, sequences of stages retrieve input and run jobs such as build, test, and deploy. The default delivery pipeline contains the components shown in Figure 4-17.

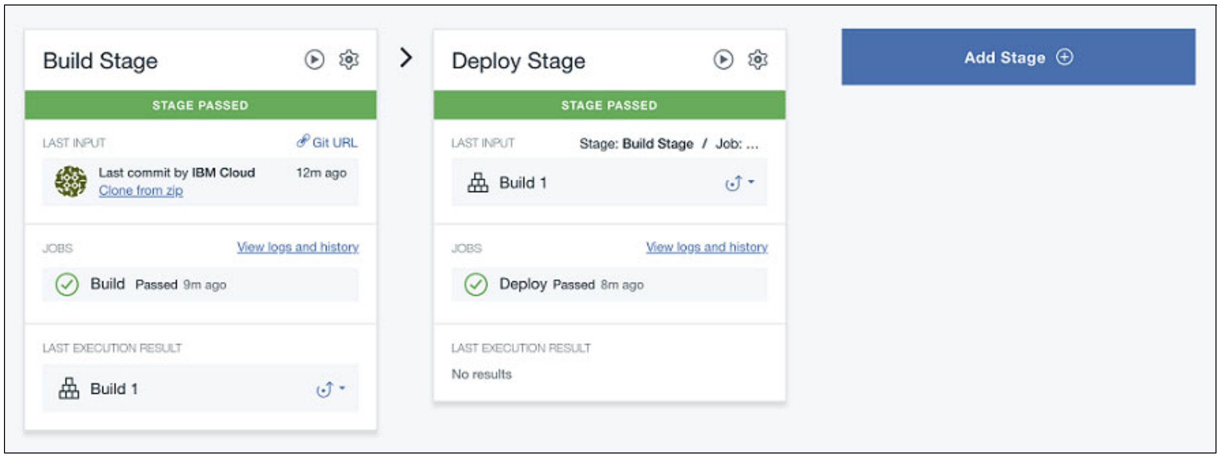
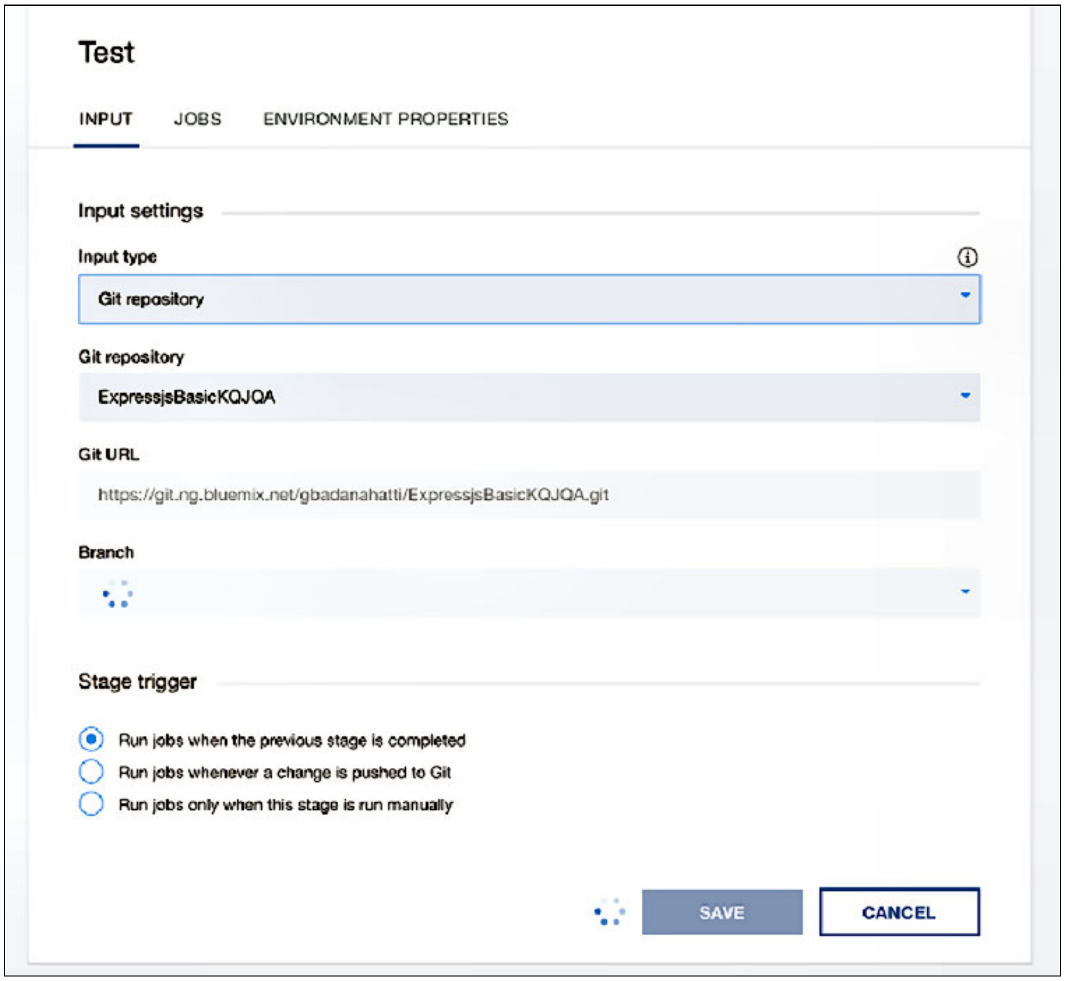


Figure 4-17 Components in the default delivery pipeline

Each stage is highly customizable, and more stages can be added to build your customized pipeline to meet your organization's needs. For example, you can add a stage for performing Unit tests, deploy it in the staging area, run more sanity tests, and then move it to production.

To add a stage that does unit tests, complete these steps:

1. Click **Add Stage**.
2. Add the details for the new stage, as shown in Figure 4-18.



Test

Input settings

Input type: Git repository

Git repository: ExpressjsBasicKQJQA

Git URL: https://git.ng.bluemix.net/gbadanahatti/ExpressjsBasicKQJQA.git

Branch

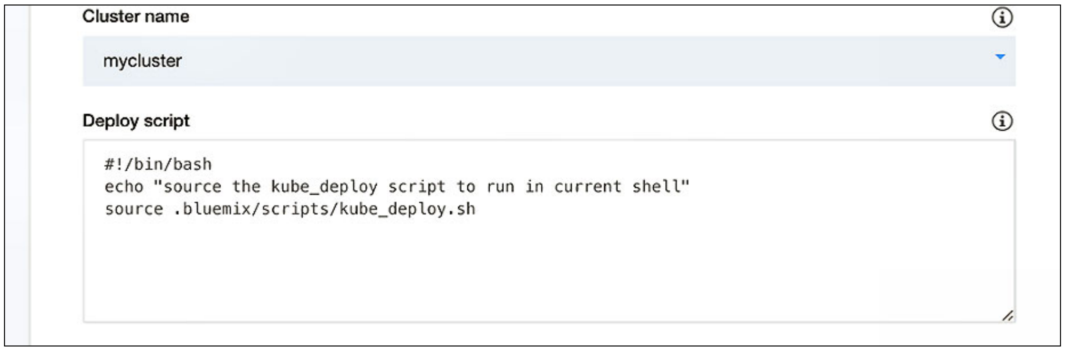
Stage trigger

(●) Run jobs when the previous stage is completed
( ) Run jobs whenever a change is pushed to Git
( ) Run jobs only when this stage is run manually

Figure 4-18 Adding details for the new stage

3. After the stage is created, drag the new tile to the correct location in the pipeline.

4. Customize the stage by adding your own Deploy script, as shown in Figure 4-19.



```
#!/bin/bash
echo "source the kube_deploy script to run in current shell"
source .bluemix/scripts/kube_deploy.sh
```

Figure 4-19   Adding a Deploy script

The pipeline configuration is one of the configuration artifacts. The Starter Kit places the Development pipeline in `.bluemix/pipeline.yml`. This file can be modified to add stages. An example of the build stage in the pipeline is shown in Example 4-1.

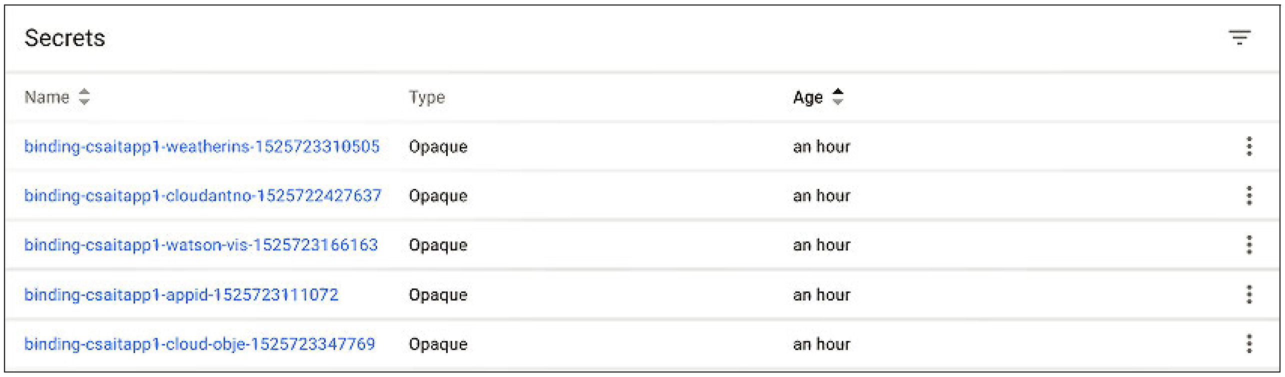
Example 4-1   Build stage example

```
stages:
- name: Build Stage
  inputs:
  - type: git
    branch: master
    service: ${REPO}
  triggers:
  - type: commit
  properties:
  - name: CHART_NAME
    value: ${CHART_NAME}
    type: text
  jobs:
  - name: Build
    type: builder
    build_type: cr
    artifact_dir: ''
    target:
      region_id: ${REGION_ID}
      api_key: ${API_KEY}
    namespace: ${REGISTRY_NAMESPACE}
    image_name: ${CHART_NAME}
    script: |-
      #!/bin/bash
   <Your script>
      fi
```

For more information, see the Delivery Pipeline web page.

4.3.4 Credential injection from Starter Kits

Every resource that is bound to the application from a Starter Kit has credentials and access points that allow it to access the service. The Starter Kit injects these attributes as Kubernetes secrets. If you navigate to the Kubernetes dashboard of your cluster (Figure 4-20) and open the Secrets window, you will see the secrets for each service.



Secrets

Name	Type	Age
binding-csaitapp1-weatherins-1525723310505	Opaque	an hour
binding-csaitapp1-cloudantno-1525722427637	Opaque	an hour
binding-csaitapp1-watson-vis-1525723166163	Opaque	an hour
binding-csaitapp1-appid-1525723111072	Opaque	an hour
binding-csaitapp1-cloud-obje-1525723347769	Opaque	an hour

Figure 4-20 The Kubernetes dashboard

The Starter Kit includes an `ibm-cloud-env` node package that abstracts environment variables from various Clouds compute providers, such as, but not limited to, Cloud Foundry and Kubernetes, so the application can be environment-agnostic.

The module allows you to define an array of search patterns that are run one by one until the required value is found. More information about this package can be found at the IBM Cloud Environment section of the NPM website.

A snippet of the `mappings.json` file is shown in Example 4-2.

Example 4-2 Portion of mappings.json

```
"cloudant_url": {
    "searchPatterns": [
      "cloudfoundry:$.cloudantNoSQLDB[0].credentials.url",
      "env:service_cloudant:$.url",
      "file:/server/localdev-config.json:$.cloudant_url"
    ]
  },
```

The access to the attributes is done in a code pattern like that shown in Example 4-3.

Example 4-3 Accessing attributes

```
const IBMCloudEnv = require("ibm-cloud-env");
const client = require("./cloudant");
const log4js = require("log4js");

const logger = log4js.getLogger("cloudant-service");

module.exports = function(app, serviceManager){
   client.init(IBMCloudEnv.getString('cloudant_url')).then(() => {
         serviceManager.set("cloudant", client);
   }).catch((err) => {
        logger.warn("Cloudant service could not be initialized: "+ err);
    });
};
```

To facilitate local development, IBM Cloud Starter Kits allow you to create a local configuration file in the location `server/localdev-config.json`. The `ibm-cloud-env` package will seamlessly pick up the `localdev-config.json` attributes during application run time, as shown in Example 4-4.

Example 4-4 Picking up the localdev-config.json attributes

```
{
    ….

"cloudant_url":
"https://xxxxx-bluemix:yyyyy@6d16902f-9774-4272-b029-03d3380fe096-bluemix.cloudant
.com",

…
}
```

4.3.5 Adding a service after deployment

When you first create a project with a starter kit you can add resources, as we did above. However, sometimes you will need to add additional services after the code has been generated and deployed. This requirement might be because you decided to add a new service later, or you needed to add a service that is not currently supported by the starter kit framework.

When you want to add a new service through the starter kit framework after deployment, you can do so in the App console. Note that adding a resource allows you to generate new code with references to the new service. However, this new code will not automatically be added to your app's code repository.

Follow the procedure below do fully integrate your new service:

1. Log in to the IBM Cloud console and click **WebApps** → **Apps**. This window lists your applications.
2. Click your application and select **Add Resource**.
3. Select the resource that you want to add and add it to your application.
4. You can now download the newly generated code into a directory.
5. Compare the changes to the `mappings.json` in the newly downloaded code bundle with what is in the git location and merge the changes to the `mappings.json` file in the git repository. A sample of a newly added Watson Conversation Service is shown in Example 4-5.

   Example 4-5 Example of an added service

   ```
   …
       "watson_conversation_url": {
       "searchPatterns": [
         "cloudfoundry:$.conversation[0].credentials.url",
         "env:service_watson_conversation:$.url",
         "file:/server/localdev-config.json:$.watson_conversation_url"
   ```

```
      ]
    },
    "watson_conversation_username": {
      "searchPatterns": [
        "cloudfoundry:$.conversation[0].credentials.username",
        "env:service_watson_conversation:$.username",
        "file:/server/localdev-config.json:$.watson_conversation_username"
      ]
    },
    "watson_conversation_password": {
      "searchPatterns": [
        "cloudfoundry:$.conversation[0].credentials.password",
        "env:service_watson_conversation:$.password",
        "file:/server/localdev-config.json:$.watson_conversation_password"
      ]
    },

---
```

6. Copy the new file in `server/services/service-<new service>.js` from the downloaded location to the git cloned location.
7. Create corresponding entries in the `localdev-config.json` to enable local development.
8. Commit and push the new changes to be deployed on the Kubernetes container.

   Continue to develop changes for the new service as needed.
9. Commit and push the new changes to be deployed on the Kubernetes container.

Note: Some services do not integrate with an existing app. If you want to add this service to the app, then you will need to complete these steps:

1. Modify the `mappings.json` file manually to create the credential mapping, like the one shown above.
2. Create a `service-<new service>.js` in the services folder using one of the existing services as a base line.
3. You might need modify the Helm charts to add any needed environment variables.

4.3.6 Adding an IBM Cloud Object Storage service after deployment

The steps below are typically used when you already have an IBM Cloud Object Storage service that you intend to use for this application rather than use the Starter Kit Wizard to create one:

1. Create an alias to the service:

   ```
   bx resource service-alias-create <alias_name> -instance-name <service name>
   ```
2. Create a service key:

   ```
   bx resource service-key-create cos-alias-service-key Writer --alias-name
   <alias_name>
   ```
3. Bind the alias to the cluster:

   ```
   bx cs cluster-service-bind <cluster name> default <alias-name>
   ```

After the execution of the above command, you will see that the credentials for IBM Cloud Object Storage will now be in the Kubernetes Secrets.

Complete the following steps to integrate the existing IBM Cloud Object Storage service into your source code:

1. Modify the `mappings.json` to add the bucket lookup information, as shown in Example 4-6.

Example 4-6 Adding the bucket lookup information

```
"cos_original_bucket": {
    "searchPatterns": [
      "env:COS_ORIGINAL_BUCKET",
      "file:/server/localdev-config.json:$.cos_original_bucket"
    ]
  },
  "cos_normalized_bucket": {
    "searchPatterns": [
      "env:COS_NORMALIZED_BUCKET",
      "file:/server/localdev-config.json:$.cos_normalized_bucket"
    ]
  },
  "cos_api_key": {
    "searchPatterns": [
      "cloudfoundry:$.cloud-object-storage.credentials.apikey",
      "env:service_cos:$.apikey",
      "file:/server/localdev-config.json:$.cos_api_key"
    ]
  },
  "cos_service_instance_id": {
    "searchPatterns": [
      "cloudfoundry:$.cloud-object-storage.credentials.resource_instance_id",
      "env:service_cos:$.resource_instance_id",
      "file:/server/localdev-config.json:$.cos_service_instance_id"
    ]
  },
  "cos_endpoint": {
    "searchPatterns": [
      "env:COS_ENDPOINT",
      "file:/server/localdev-config.json:$.cos_endpoint"
    ]
  },
  "cos_ibm_auth_endpoint": {
    "searchPatterns": [
      "env:COS_IBM_AUTH_ENDPOINT",
      "file:/server/localdev-config.json:$.cos_ibm_auth_endpoint"
    ]
  },
```

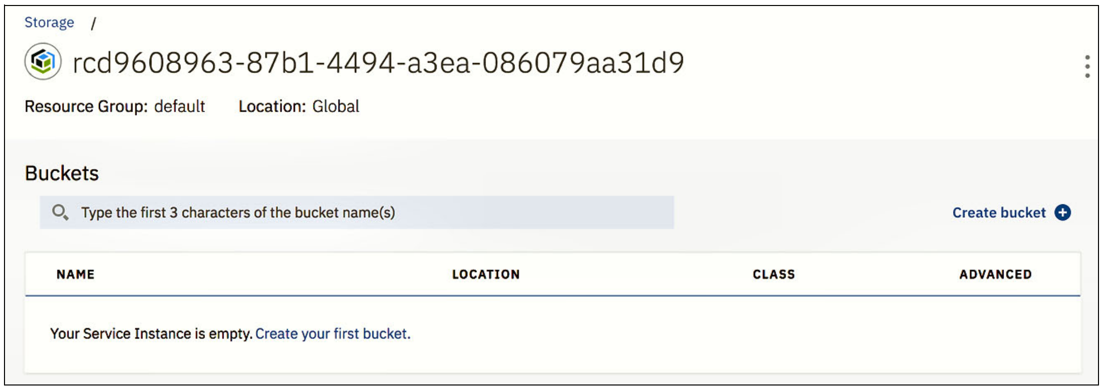
2. Modify the Helm charts to add any needed environment variables. Add the code shown in Example 4-7 in the services section of `values.yaml` file in `chart/<appname>`.

Example 4-7 Modifying the Helms charts

```
services:
  cos:
    originalBucket: claims-images-original
    normalizedBucket: claims-images-normalized
    endpoint: s3.us-south.objectstorage.softlayer.net
ibmAuthEndpoint: https://iam.bluemix.net/oidc/token
```

4.3.7 Bucket name injection to application

The Starter Kit will create an Object Storage Instance, but you need to create the buckets by using either IBM Cloud Object Storage API or the console, as shown in Figure 4-21.


Storage /

rcd9608963-87b1-4494-a3ea-086079aa31d9

Resource Group: default   Location: Global

Buckets

Type the first 3 characters of the bucket name(s)

Create bucket

NAME	LOCATION	CLASS	ADVANCED

Your Service Instance is empty. Create your first bucket.

Figure 4-21   Creating buckets

Click **Create bucket** and create two buckets: One bucket for holding original uploaded image and the other to hold the normalized images. The bucket names are made known to the cluster through the Kubernetes environment variables. To accomplish this, place the bucket names in the helm chart files located in `charts/values.yaml` and `charts/deployment.yaml`.

Note: If you want key-based protection using key protect, ensure that you add the keys during the bucket creation. Key protect services cannot be added after a bucket has been created.

Example 4-8 shows the contents of `charts.yaml`.

Example 4-8   charts.yaml

```
…
services:
  cos:
    originalBucket: claims-images-original
    normalizedBucket: claims-images-normalized
…
```

Example 4-9 shows the contents of `deployment.yaml`.

Example 4-9   deployment.yaml

```
…
        env:
          - name: COS_ORIGINAL_BUCKET
            value: "{{ .Values.services.cos.originalBucket }}"
          - name: COS_NORMALIZED_BUCKET
            value: "{{ .Values.services.cos.normalizedBucket }}"
…
```

Bucket names to which images are written to are injected to the application with the `mappings.json` file and read through the `ibm-cloud-env` modules, as shown in Example 4-10.

Example 4-10 Bucket names

```
{
…

  "cos_original_bucket": {
    "searchPatterns": [
      "env:COS_ORIGINAL_BUCKET",
      "file:/server/localdev-config.json:$.cos_original_bucket"
    ]
  },
  "cos_normalized_bucket": {
    "searchPatterns": [
      "env:COS_NORMALIZED_BUCKET",
      "file:/server/localdev-config.json:$.cos_normalized_bucket"
    ]
  }
…
}


const IBMCloudEnv = require('ibm-cloud-env');

// Bucket names - taken from environment variables set in Helm chart
const _originalBucket = IBMCloudEnv.getString('cos_original_bucket');
const _normalizedBucket = IBMCloudEnv.getString('cos_normalized_bucket');
```

4.3.8 User interface components

Now that the application has been enabled, add some user interface components to enable you to visualize the modules whose details you will dive into shortly:

1. Create an `index.html` page, as shown in Figure 4-22. This will be the landing page of your content. The landing page prompts you to log in to the application. See the `index.html` in public folder of the git repository for more details.

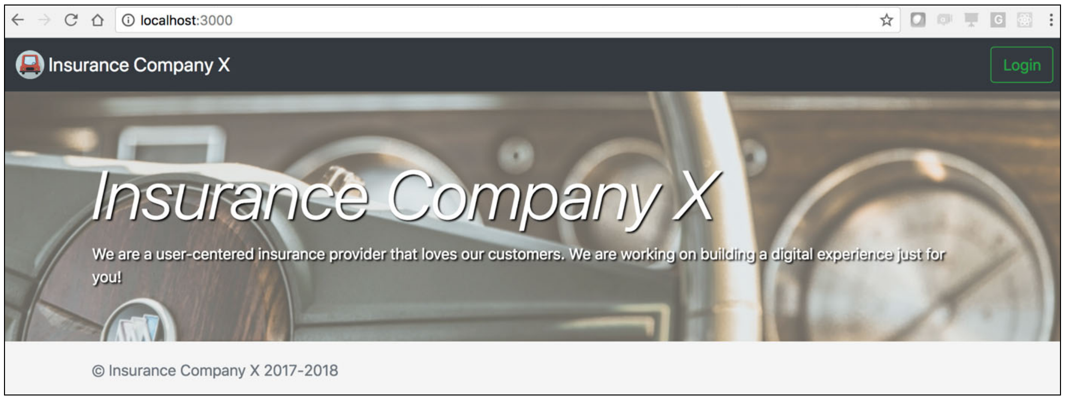
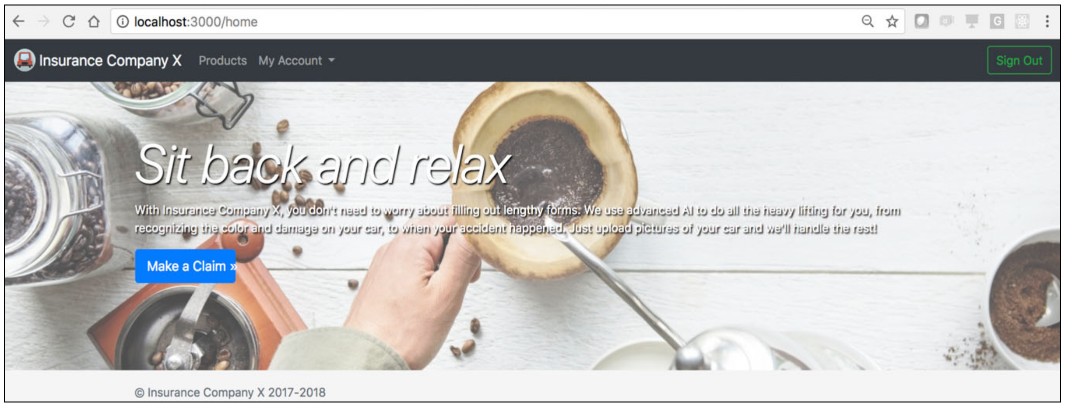

localhost:3000
Insurance Company X
Login
Insurance Company X
We are a user-centered insurance provider that loves our customers. We are working on building a digital experience just for you!
© Insurance Company X 2017-2018

Figure 4-22 Creating an index.html page

After clicking **Login**, the user is authenticated and is redirected to `home.html` page.

2. Create a `home.html` page. This is the page that the user will be directed to after a successful login. This page contains all the controls for invoking actions on the web application. See `home.html` in the public folder of the git repository for more details. The UI is shown in Figure 4-23.



Sit back and relax

With Insurance Company X, you don't need to worry about filling out lengthy forms. We use advanced AI to do all the heavy lifting for you, from recognizing the color and damage on your car, to when your accident happened. Just upload pictures of your car and we'll handle the rest!

© Insurance Company X 2017-2018

Figure 4-23 UI home page

3. In this module, we introduce a skeleton claims processing. To make a claim, the user clicks **Make a Claim** in the user interface. This action invokes the new`ClaimsBtn` Handler in the `action.js` JavaScript file located in `js/actions.js`, as shown in Example 4-11.

Example 4-11 Making a claim

```
$("#newClaimBtn").click(function(e) {
    console.log("Getting new claim");
    e.preventDefault();

    // create a claim to start working on
    $.post(
      "/claim",
      {},
      function(data, textStatus, jqXHR) {
        console.log (textStatus);
        if (textStatus == "success") {
          var claimFQDN = jqXHR.getResponseHeader("Location");
          var claimId = claimFQDN.split("/")[2];
          var fullClaimLocation = jqXHR.getResponseHeader("Location");
          console.log('found claim location', fullClaimLocation);

          // Now render the page
          uiDisplayEditClaim(claimId);
        } else {
          alert("Failed to create claim!");
        }
    });
  });
```

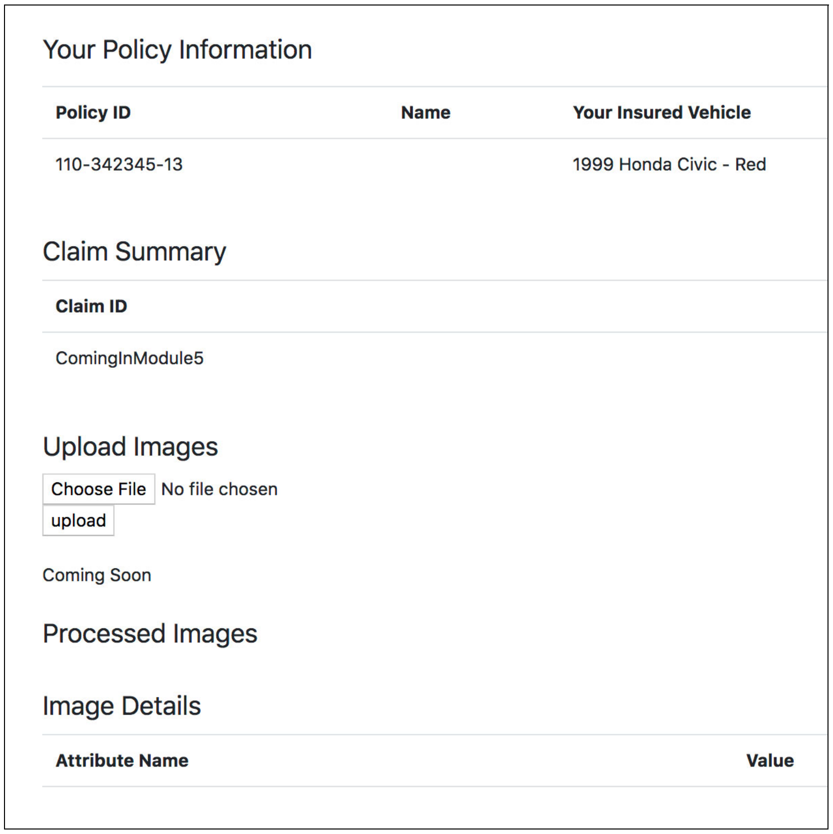
4. This process ends a POST request to the server that sends a dummy response back to the client. This process is handled on the server side in the file `index.js` in the `server/claims` directory, as shown in Example 4-12.

Example 4-12 Handling the POST request

```
router.post("/", function(req, res) {
                        res.setHeader("Location", `/claim/ComingInModule5`);
                        res.status(201).end();
        });

        // GET a list of claims
        router.get("/", function (req, res) {
                        var claims = {'claims': 'ComingInModule5'};
                        res.json(JSON.stringify(claims));
        });
```

The user interface shown in Figure 4-24 is displayed to the user when the claim is handled by the server.


Your Policy Information

Policy ID	Name	Your Insured Vehicle
110-342345-13		1999 Honda Civic - Red

Claim Summary

Claim ID
ComingInModule5

Upload Images

Coming Soon

Processed Images

Image Details

Attribute Name	Value

Figure 4-24 Claim user interface

Module 5 expands on this skeleton code to add business logic.

4.4 Next steps

In the next module, you will enhance the Insurance Company X's Web Application to create and configure an instance of a Cloudant to store the metadata and learn more about Cloudant.

4.5 Other references

- IBM Cloud App Service Starter Kits Documentation

  `https://console.bluemix.net/developer/appservice/documentation`

5

Storing metadata in Cloudant

Now that you have completed Module 4, you have a first version of the claims application running. This module covers how you persist information. It focuses on using IBM Cloudant to store all the metadata that is associated with a claim.

Metadata is often managed in a traditional relational database management system (RDBMS) like Postgres. Although this technique provides solid capabilities for managing data, it is challenging to distribute an RDBMS to multiple regions, scale it dynamically and seamlessly, and evolve the structure of the data. However, these are core requirements for a claims application.

By using IBM Cloudant, you can implement an elastic and geographically dispersed database for claims, while not having to worry about deploying, scaling, or managing the database. In addition, the schema-free nature allows you to change the structure of the data easily, resulting in faster time-to-market and lower cost.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

5.1 Learning objectives

After completing this module, you will be able to:

- Create and configure an instance of Cloudant.
- Describe the differences between Cloudant and other database offerings.
- Implement code to read, write, and index data.
- Describe the different ways to work with data in Cloudant.
- Identify the best way to structure data for a specific use case.

5.2 Getting started

To adopt the code to your account and application, you must customize some files after checking out the code and before adding the changes discussed in this chapter:

1. Clone the git repository that is linked to the continuous integration pipeline that you created in the previous chapter:

   ```
   $ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
   $ cd IBM-Cloud-Object-Storage-Tutorials
   ```

2. Copy all code excluding the `.git` folder from GitHub into your repository.
3. Open `chart/insurancewebappbackend/values.yaml` and adjust the repository to match your docker repository.
4. Open `chart/insurancewebappbackend/deployment.yaml` and replace the secret key reference binding for `service_cloudant` and all other services with the ones found in your chart's `deployment.yaml`.
5. Rename the helm chart under chart from `insurancewebappbackend` to the name of your application and overwrite your original chart.
6. Open `chart.yaml` and change the value of the name property.
7. Commit the changes.

5.3 Implementation

Besides claims, we will store some auxiliary information like accounts and policies. Claims make up most of the data that is to be persistent. A Claim consists at least of this data:

- Claim identifier
- State of the claim: Open, approved, archived
- Image storage location in IBM Cloud Object Storage

Beyond this data, every additional service integration can contribute additional metadata to a claim. Examples include location, weather, and classification information such as car color and damage type. Because of this highly dynamic nature, a schema free database seems most suitable for our needs.

Another aspect that you must take into account is the need to run in multiple geographically dispersed locations in an active-active setup.

5.3.1 IBM Cloud persistency options

Each database can be categorized by using the CAP-Theorem to help match the specific service against the needs of your application.

The CAP-Theorem as postulated by Eric Brewer in 2000 states that because network failure is inevitable in a distributed system, you can at most have two out of three properties in such a system. The following are the three properties:

- *Consistency* ensures that when a transaction completes, all replicas of the data are ensured to be consistent. Therefore, each client always has the same view of the data. This concept is not to be confused with consistency as implied by ACID, which employs a different definition of consistency.
- *Availability* is the ability to respond to a request within an acceptable amount of time.
- *Partition tolerance* is a measure for how the system keeps working after partial loss of connectivity, nodes, or data.

Figure 5-1 shows the balance diagram.

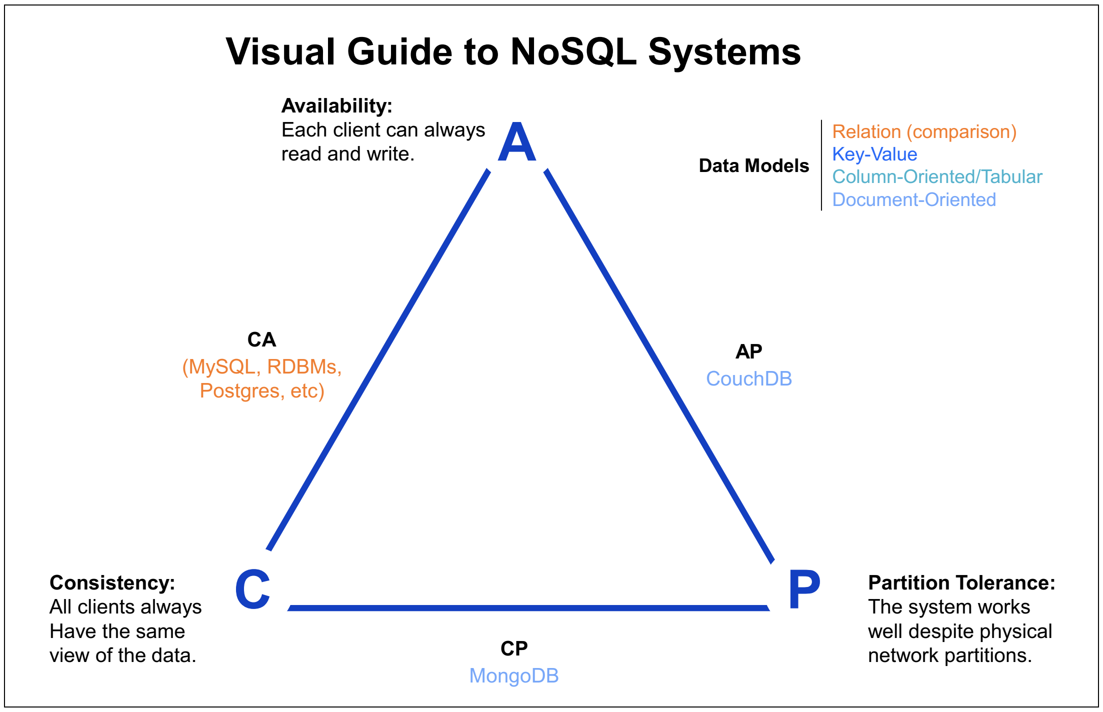

Visual Guide to NoSQL Systems
Availability:
Each client can always read and write.
A
Data Models
Relation (comparison)
Key-Value
Column-Oriented/Tabular
Document-Oriented
CA
(MySQL, RDBMs, Postgres, etc)
AP
CouchDB
Consistency:
All clients always Have the same view of the data.
C
P
Partition Tolerance:
The system works well despite physical network partitions.
CP
MongoDB

Figure 5-1 Visual guide to NoSQL systems

Postgres is an example of an RDBMS, and therefore prefers consistency and availability over partition tolerance. This choice means that a cluster cannot span multiple regions because distance limits throughput and the loss of one region will bring Postgres down.

MongoDB, on the other hand, easily scales to a large set of nodes. MongoDB implementation prefers consistency over availability. This goal is achieved by defining a group of nodes that host the same data set as a replica set. Within each group, a primary node takes care of all write operations and by default also read operations. Replication is used to recover from hardware failure or service interruption. When the primary node fails, the remaining secondaries run an election and promote a new primary through quorum. During that time, the data cannot be read or written. If no quorum can be achieved, the election fails, and the replica set remains unavailable.

Cloudant shares some characteristics with MongoDB, including easily scaling to a large set of nodes. However, the underlying CouchDB prefers availability over consistency. This goal is achieved by using a replication model called Eventual Consistency. In this system, clients can write to any node of the database without waiting for other nodes to reach quorum about the write. Nodes incrementally copy changes between nodes, ensuring that all nodes will eventually be in sync.

Because there is no single point of failure in this system, it is highly available and can survive the failure of multiple nodes. Furthermore, this model also allows for cross-cluster replication, which helps to easily spread the database into multiple regions without decreasing availability. The price that you must pay for this advantage is that the application needs to account for the fact that data might not be consistent.

One of the objectives for the application is to deploy into multiple regions. Therefore, we have selected Cloudant to store Claim and Accounts data. Using a document-oriented JSON database also has the benefit that you do not need to migrate data or change the schema when adding data to a claim.

Now that we have decided to use Cloudant, design your database schema. Unlike in RDBMS, data should be highly denormalized because cross references between documents cause consistency issues. Taking the eventual consistency aspect into account, the preferred pattern is to never update documents, but only write new documents. This technique effectively turns the database into an event-based change feed. Because Claims follow several legal and audit regulations, this is actually a very useful property of this data model. Using a write only model ensures a complete and consistent audit trail for every claim.

For a more in-depth discussion about how to properly design your data model for Cloudant, see Cloudant Best (and Worst) Practices.

Using Cloud Object Storage with Cloudant: While Cloudant supports binary data such as claim images, inline within the database, the recommended approach, which we implement in this application, is to store binary data on Cloud Object Storage and only keep a reference to this data in Cloudant.

Attachments are not included in Cloudant backups and there can be a negative performance impact when storing many large objects in a database. Storing binary data in Cloud Object storage provides better resiliency through cross regional Cloud Object Storage at lower cost and enhances the replication performance of Cloudant.

Working with a database

The client libraries for Cloudant were added when the Starter Kit was generated, which means that you can start interacting with the database right away.

Methods to interact with Cloudant, to create databases, create indices, insert design documents, and query the database will be added in `server/services/cloudant/index.js`. This method helps to isolate Cloudant specific logic from the rest of the application. First, add all necessary module imports as well as globals and the exported method, as shown in Example 5-1.

Example 5-1 Adding all module imports

```
"use strict";

// Modules
const Cloudant = require("@cloudant/cloudant");
const log4js = require("log4js");
const uuid = require("uuid/v4");
const moment = require("moment");
const __ = require("lodash");

// Globals
let db;
const DB_NAME = "car_claims";
const CLAIM_INDEX = require("./claimIndex.json");
const logger = log4js.getLogger("cloudant-client");

// Public Methods --------------------------------------------------------------->

module.exports.init = _init;
//module.exports.createClaim = _createClaim;
//module.exports.appendClaim = _appendClaim;
//module.exports.getClaim = _getClaim;
//module.exports.listAllClaims = _listAllClaims;
//module.exports.archiveClaim = _archiveClaim;
```

Ensure that the database is created when your application starts, as shown in Example 5-2.

Example 5-2 Creating the database

```
function _init(cloudant_url) {
   if (!cloudant_url) {
      return Promise.reject("Cloudant configuration missing");
   }
   const cloudant = new Cloudant({ url: cloudant_url, plugins: [ "promises"] });

   // list all databases and create ours if it's not present
   return cloudant.db.list().then((db) => {
      const found = db && db.indexOf(DB_NAME) !== -1;
      if (!found) {
         logger.debug(`Database ${DB_NAME} does not exist, creating it`);
         return cloudant.db.create(DB_NAME);
      }
   });
}
```

The code lists the names of all databases, creating the database if it does not exist. When you start the application, you should see that the database is created in the Cloudant Dashboard. You can find the dashboard on the Service Instance tab on your IBM Cloud console.

After completing this initial setup, add the different functions for creating, updating, and reading claims.

Creating a claim

Add your first document by creating a function to create a claim, as shown in Example 5-3.

Example 5-3 Function to create a claim

```
function _createClaim(claimDetails) {
   // create the document from provided details, generated UUID and created date
   const document = Object.assign({},claimDetails, {type: "claim", claimId:
uuid(), created: moment.utc()  });
```

```
   // just in case someone tried to set the id
   delete document._id;
   // insert the document and return the claimId
   return db.insert(document).then((d) => {
      logger.debug(d);
      return document.claimId;
   });
}
```

Because Cloudant natively stores JSON documents, you can easily pass any object to the library to store its content. In addition to the claim data passed by the caller, the code adds these mandatory properties for every claim object:

- Type allows you to distinguish different document types (Claim/Account).
- Claim ID is the unique ID of this claim. All documents with the same Claim ID are considered part of the same claim.
- Created indicates the time that the claim was created and allows you to chronologically order documents that belong to the same claim.

The Claim ID is returned to the caller after the document has been stored in Cloudant.

Now expose this function on a route so that you can create a claim by issuing a POST against `/claim`. Complete these steps:

1. Set up an express router and group all code related to the `/claim` route under `/routers/claims`.
2. Because all code on the `/claim` route needs access to Cloudant, add middleware in `/routers/common.js` that adds the Cloudant service to the request object. This configuration allows it to be easily used by downstream code so you do not need to repeat the same code. In cases where Cloudant is not configured properly, the request fails and returns an error. Example 5-4 shows the middleware.

   Example 5-4 Middleware to add Cloudant service

   ```
   module.exports.cloudantInjector = function (req, res, next) {
      req.cloudant = serviceManager.get("cloudant");
      if (!req.cloudant) {
         res.status(500).send("Cloudant not configured, cannot search!");
      }
      next();
   };
   ```

3. Having laid the foundation for all functions you will add to the claims route, add a method to create a Claim. Example 5-5 shows the Claim creation code.

   Example 5-5 Code to create a Claim

   ```
   module.exports = function(app) {
      const router = express.Router();

      router.post("/", function(req, res) {
         req.cloudant.createClaim(req.body).then((claimId) => {
            // Tell caller the location of the created resource
            res.setHeader("Location", `/claim/${claimId}`);
            res.status(201).end();
         }).catch((err) => {
            logger.error(err);
   ```

```
            res.status(500).send("Something broke!");
         });
      });
   // Register the route /claim
      app.use("/claim",
         bodyParser.json(),
         common.cloudantInjector,
         router);
   }
```

createClaim returns a Claim ID that is sent back in the location header to indicate the location of the newly created claim to the caller.

Updating a Claim

Go back to server/services/cloudant/index.js to add a function to append data for a claim.

As you will not update a document in Cloudant, all documents belonging to the same claim need to share the claim ID as a common property to allow you to group them together, as shown in Example 5-6.

Example 5-6 Appending data to a Claim

```
function _appendClaim(claimAppend) {
   // enforce presence of claimId, as this is the field that we use to aggregate
   if (!claimAppend.claimId) {
      return Promise.reject("Missing claimId parameter!");
   }

   // make sure the document has a proper created timestamp
   const document = Object.assign({},claimAppend, {created: moment.utc(), type:
"claim"});
   delete document._id;
   // insert the document
   return db.insert(document).then((d) => {
      logger.debug(d);
      return;
   });
}
```

The code for updating a claim is similar to creation of a claim, with the exception that a claim ID must be provided in this case.

To enable updates of claims through REST, add a function that responds to a patch against a specific claim ID in /routers/claims/index.js, as shown in Example 5-7.

Example 5-7 Function to respond to patch

```
router.patch("/:claimId", function(req, res) {
  const append = Object.assign({},req.body,{claimId: req.params.claimId});

  req.cloudant.appendClaim(append).then(() => {
    res.status(200).end();
  }).catch((err) => {
    logger.error(err);
    res.status(500).send("Something broke!");
  });
});
```

The claim update is augmented with a created time, a type property, and the claim ID to ensure that the documents fulfill the requirements of the data model.

Fetching claim details

Getting all the details for a claim requires the system to read multiple documents from Cloudant. You must find all documents that belong to the same Claim. To hide the details of the write-only data model from the user, fuse all the documents for a claim into a single view.

Because an update to a claim might overwrite a previous value, documents need to be ordered by their creation time to merge them correctly.

Enabling querying and sorting by Claim ID and created time requires you to create an Index for these fields in addition to the document type, as shown in Example 5-8.

Example 5-8 Creating an Index

```
{
  "name": "claimIndex",
  "ddoc": "claimIndex",
  "type": "json",
  "index": {
  "fields": [
     {
       "type" : "asc"
     },
    {
     "claimId": "asc"
    },
    {
     "created": "asc"
    }
   ]
  }
}
```

Adding the index to Cloudant is best done after validating that the database exists in `server/services/cloudant/index.js`, as shown in Example 5-9.

Example 5-9 Validating that the database exists

```
const CLAIM_INDEX = require("./claimIndex.json");
...
function _init(cloudant_url) {
   if (!cloudant_url) {
      return Promise.reject("Cloudant configuration missing");
   }
   const cloudant = new Cloudant({ url: cloudant_url, plugins: [ "promises"] });

   // list all databases and create ours if it's not present
   return cloudant.db.list().then((db) => {
      const found = db && db.indexOf(DB_NAME) !== -1;
      if (!found) {
         logger.debug(`Database ${DB_NAME} does not exist, creating it`);
         return cloudant.db.create(DB_NAME);
      }
   }).then(() => {
      logger.debug(`Using database ${DB_NAME}`);
      db = cloudant.db.use(DB_NAME);
   }).then(() => db.index(CLAIM_INDEX)); // create the claim index
}
```

Now that you have enabled searching on these fields in the database, add a method that will fetch all documents for a claim, as shown in Example 5-10.

Example 5-10 Fetching all documents for a claim

```
function _getClaim(claimId) {
   if (!claimId) {
      return Promise.reject("Missing claimId parameter");
   }
   // Leverage the index we created to sort by claimId and created date
   const query = {
      "selector": {
         "type" : "claim",
         "claimId": claimId
      },
      "sort": [
         {
            "type" : "asc"
         },
         {
            "claimId": "asc"
         },
         {
            "created": "asc"
         }
      ]
   };

   // Search all documents for this claim
   return db.find(query).then((docs) =>  {
      // if there are none, return null
      if (!docs || docs.docs.length === 0) {
         return null;
      }

      logger.trace(`Will merge ${docs.docs.length} documents for case`);
      // First, remove all internal properties starting with _, then merge all 
documents
      const cleanedDoc = __.map(docs.docs, (doc) => __.pickBy(doc, (v,k) => 
!__.startsWith(k,"_")));
      const unionDocument = __.merge({},...cleanedDoc);
      // created dates of states have a semantically different meaning for the 
overall case
      unionDocument.created = docs.docs[0].created;
      unionDocument.lastUpdate = docs.docs[docs.docs.length - 1].created;
```

```
    return unionDocument;
  });
}
```

Using the Cloudant query syntax, retrieve all documents of type claim for a specific claim ID and ensure that the results are ordered by the created date.

A couple of system properties like `_id` and `_rev` have been added by Cloudant to each document. Because you do not want to return these properties, first remove all properties that start with underscore from each document. Then, merge the documents in the order that they were created. This technique ensures that later updates overwrite earlier values.

The merge implementation recursively merges all objects and arrays to produce the final document.

Finally, use the creation time of the first and last document to indicate creation time and last update time for the claim to the caller.

To make this data available in the REST API, add code to inject the current claim by using express middleware similar to how you injected Cloudant to the request object, as shown in Example 5-11.

Example 5-11 Middleware to inject Cloudant into the request object

```
// Ensures claim exist and attaches it to the request
const claimInjector = function (req, res, next) {
  // get the claimId from the request parameters
  const claimId = req.params.claimId;

  req.cloudant.getClaim(claimId).then((claimDetails) => {
    if (claimDetails) {
      req.claim = claimDetails;
      next();
    } else {
      res.status(404).send("Claim does not exist");
    }
  }).catch((err) => {
    logger.error(err);
    res.status(500).send("Something broke!");
  });
};
```

Having that function in place makes the actual implementation of a get for a specific claim trivial, as shown in Example 5-12.

Example 5-12 Implementing a GET

```
// GET the details of a claim by :claimId
router.get("/:claimId", claimInjector, function (req, res) {
  res.json(req.claim);
});
```

Along those lines, implement a function to retrieve the complete list of all claims, which mainly differs in not restricting the claim ID, as shown in Example 5-13.

Example 5-13 Function to retrieve claims

```
function _listAllClaims() {
   // leverage the index for sorting, but get all documents
   const query = {
      "selector": {
         "type" : "claim"
      },
      "sort": [
         {
            "type" : "asc"
         },
         {
            "claimId": "asc"
         },
         {
            "created": "asc"
         }
      ]
   };

   return db.find(query).then((docs) =>  {
      logger.trace(`Found ${docs.docs.length} cases`);
      // First, remove all except caseId, then make things unique
      const cleanedDocs = __.map(docs.docs, (doc) => __.pickBy(doc, (v,k) =>
!__.startsWith(k,"_")));
      // Second, group by id so that we can merge the changesets
      const groupedDocs = __.groupBy(cleanedDocs, (doc) => doc.claimId);
      // Finally, perform the merge
      const unionDocs = __.map(groupedDocs, (docs) => __.merge({},...docs));
      return unionDocs;
   });
}
```

Backup

Although Cloudant clusters are generally backed up for disaster recovery, create regular backups of your database so you can restore from them in case of failures in application logic or human error.

To create these backups, implement active-active replication and backups by completing the steps outlined in the Cloudant Backup and Recovery guide.

Wiring the user interface to the REST API

Now that you have enabled the REST API to create and update claims, link the user interface to the API so that the Claim ID of the created claim is displayed properly.

Open `public/js/actions.js` and add the code necessary to create a table containing the claim ID you just received from the API in line 12, as shown in Example 5-14.

Example 5-14 Creating a table

```
$.getJSON('/claim', function(data) {
       if (data) {
        $.each(data, function(i, claim) {
          var table_entry = '<tr><td>';
          table_entry += '</td>';
          table_entry += '<td>' + claim.claimId + '</td><td>' + claim.created + 
'</td>';
          table_entry += '<td><a href="#" onclick="navEditClaim(\'' + 
claim.claimId + '\')">Edit Claim</a></td></tr>';
          claim_rows += table_entry;
        });
        document.getElementById('allClaimsTableBody').innerHTML=claim_rows;

        uiDisplayAllClaims();
       } else {
         alert('Could not load your claims!');
       }
     }
}
```

5.4 Next steps

In the next modules, you will add more code to display images and other information that you extract, such as weather data. The next module is Chapter 6, "Using IBM Cloud Object Storage" on page 63.

5.5 Other references

- 10 Common Misconceptions about CouchDB

  `https://www.youtube.com/watch?v=BKQ9kXKoHS8`
- Cloudant Fundamentals: The Document

  `https://medium.com/ibm-watson-data-lab/cloudant-fundamentals-the-document-855c5ab92051`

6

Using IBM Cloud Object Storage

Part six of this tutorial series shows you how to build an application for Insurance Company X. This part focuses on using IBM Cloud Object Storage to store images that customers have taken of their cars. Later modules show you how to do exciting things with those images such as classifying them with Visual Recognition and extracting the metadata to determine the weather in the location where they were taken.

Imagine that you leave work only to discover that someone has crashed into your parked car, leaving a large dent in the rear bumper. You just have to snap a few pictures of the damage and upload them, and the insurance company takes care of the rest.

But what happens to all those photos after they have been uploaded? For many companies, managing huge volumes of media such as images, audio, and video is costly. Ensuring availability and resiliency of that data is a large challenge. By using IBM Cloud Object Storage, you let IBM take care of managing the infrastructure while you focus on developing services that your customers will love. This technique results in faster time-to-market and improved reliability, durability, and security of your most important data

Figure 6-1 shows a diagram of this technique.

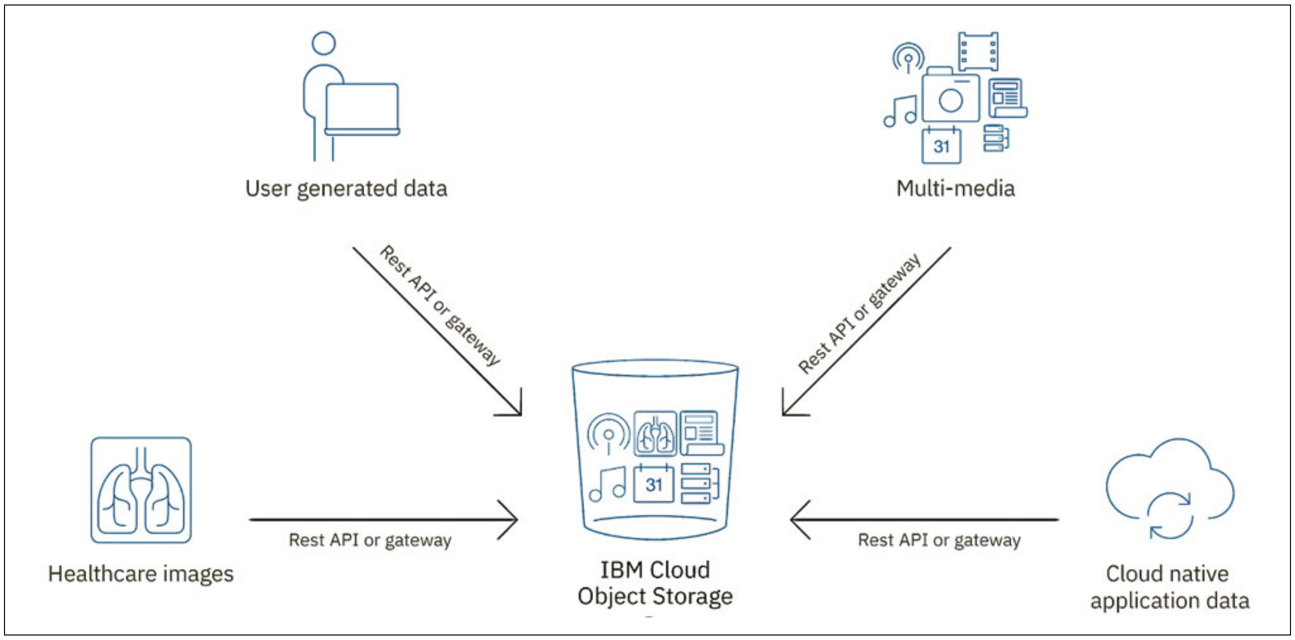


Figure 6-1 Cloud management diagram

You can use analytics on the data that you collect to gain insights into your customers so that you can offer a more personalized service. Tapping into structured and unstructured data sources using platforms such as Apache Spark can yield significant business benefits such as reducing customer churn. IBM Cloud Object Storage is a low-cost, scalable data layer that is optimized for use with Spark. Furthermore, advancements in machine learning and AI mean that more insight can be generated from this unstructured data than ever before. However, accurate machine learning models require large amounts of training data. Using IBM Cloud Object Storage is a cost-effective option that is already integrated into Watson Studio.

Figure 6-2 shows the integration of IBM Cloud Object Storage.

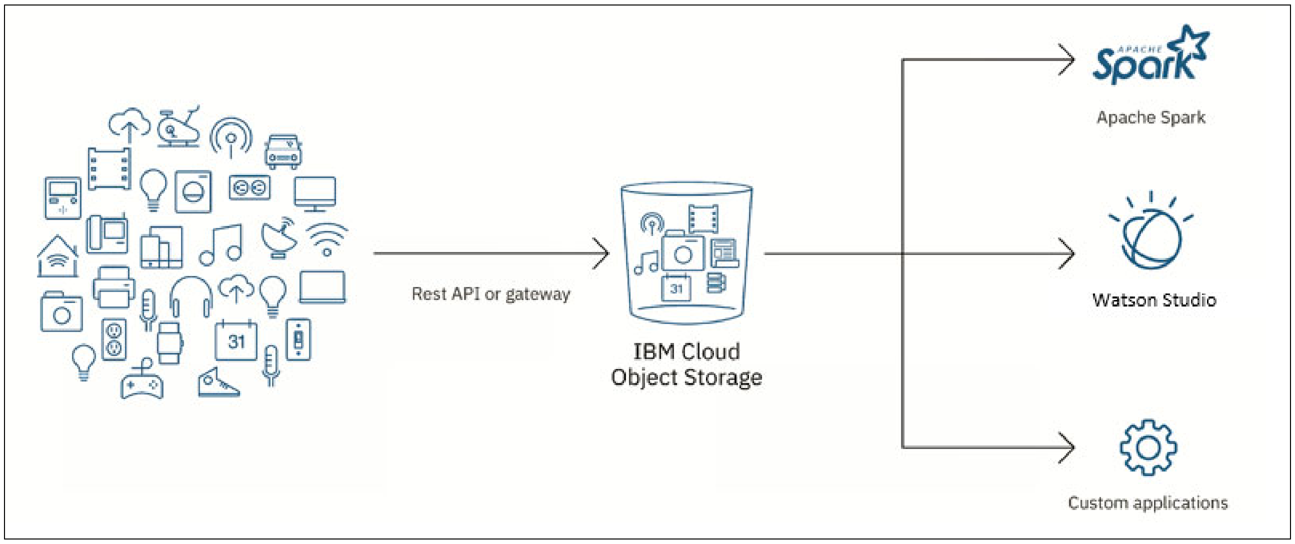


Figure 6-2 IBM Cloud Object Storage integration

This chapter includes the following sections:

- Learning objectives
- Getting started
- Configuration
- Implementation
- Next steps
- Other references

6.1 Learning objectives

After completing this module, you will be able to:

- Implement code to read from and write data to IBM Cloud Object Storage.
- Describe how to use IBM Cloud Object Storage within a broader application.
- Identify the difference between single-part and multi-part uploads.

6.2 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 5 to get the finished code from the previous module that you will build on:

```
$ cd module05
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module06
```

6.3 Configuration

If you want to run this application locally on your laptop, you must enter the credentials for the Cloudant and IBM Cloud Object Storage service in `server/localdev-config.json`. You need to enter the following credentials:

- `cloudant_url`: The Cloudant URL found in the credentials section of your Web App starter kit
- `cos_original_bucket`: The name of the bucket you created in Module 4 for the original images
- `cos_normalized_bucket`: The name of the bucket you created in Module 4 for normalized images

6.4 Implementation

This module covers the different ways to read and write data to Cloud Object Storage and why you might choose one way over another. It also outlines any setup or installation that you need to do and provides code examples showing how we are reading and writing data in the Insurance Company X application.

Because you are writing a Node.js application, it makes sense to use the IBM Cloud Object Storage Node.js SDK because it gives you an easier way to interact with IBM Cloud Object Storage in your programming language of choice. This module focuses on using the Node.js SDK to read and write data. It also describes preferred practices when using IBM Cloud Object Storage in an application and provides some code to show how we have implemented those practices in the Insurance Company X application.

So far in this series, you have created a basic app using the Starter Kits and added a basic UI, REST API, and data persistence using Cloudant. Now you will implement an integration with IBM Cloud Object Storage. Users of the application will upload photos of their damaged cars after an accident. You need to store these photos so they can be analyzed, displayed in the UI, and stored for regulatory and audit purposes. Because you have different needs for the images, store two copies. Upload the original photo, but also normalize the image to reduce the file size and get a consistent resolution for machine learning analysis.

Figure 6-3 shows what the application architecture will look after you have completed this module.

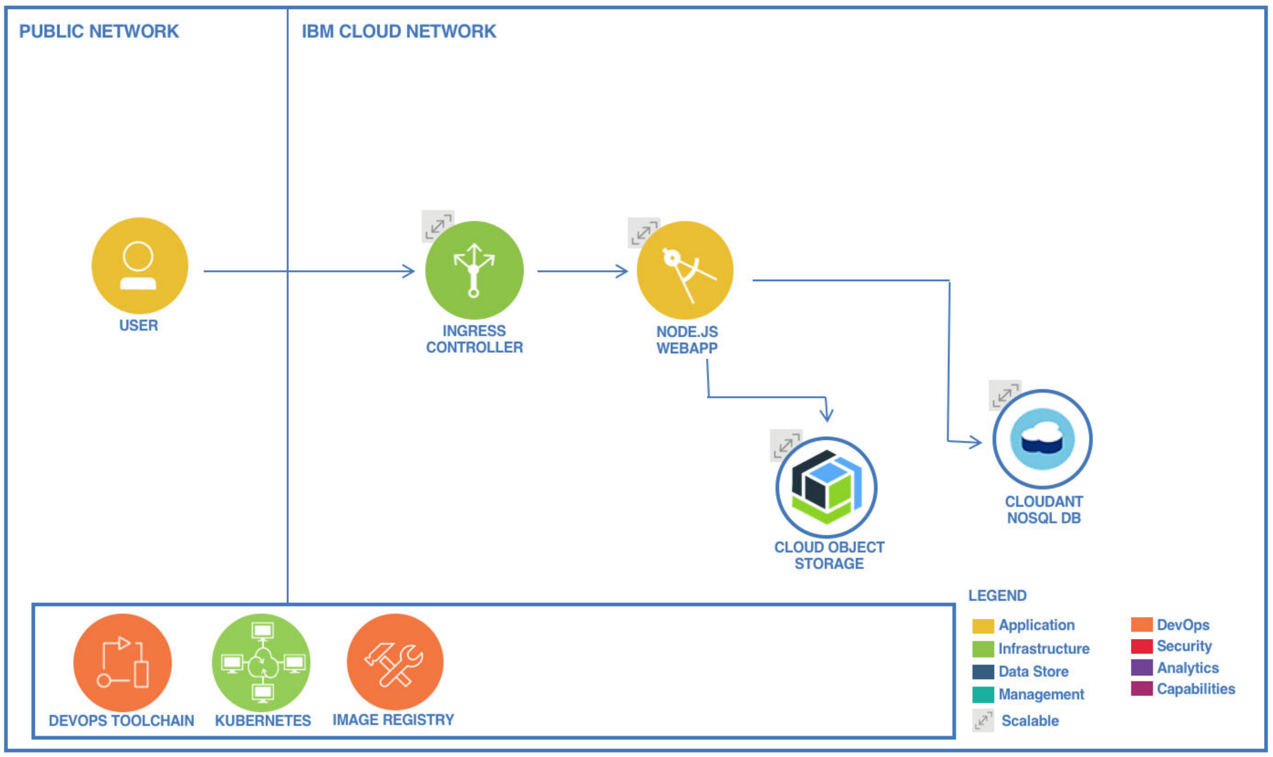


Figure 6-3 Application architecture

6.4.1 Step-by-step

You should already have created an IBM Cloud Object Storage service instance as part of the application scaffolding in Chapter 4, “Scaffolding using a Starter Kit” on page 29.

Complete the following steps:

1. Go to the IBM Cloud dashboard by clicking the menu at the upper left of the window and selecting **Dashboard**, as shown in Figure 6-4.

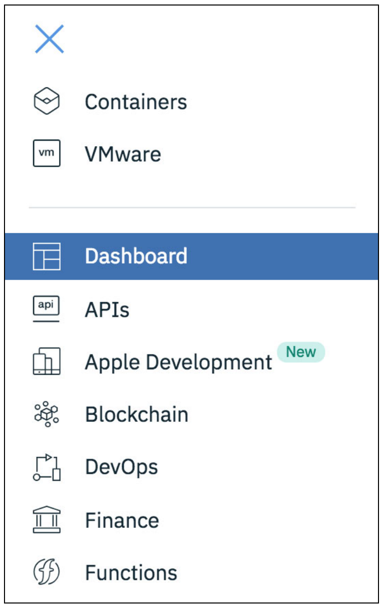
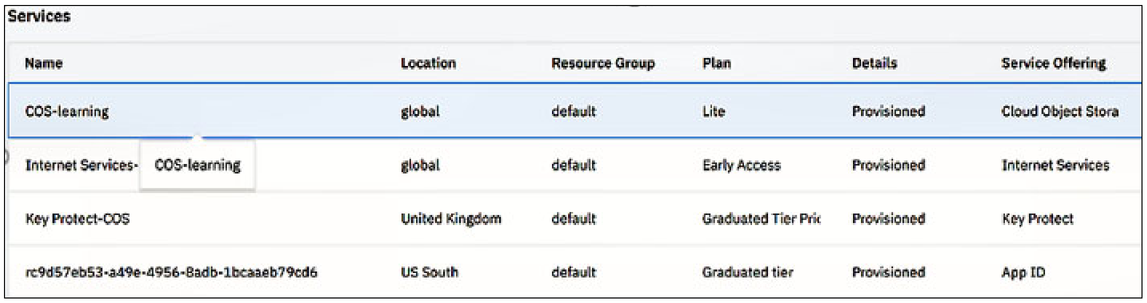


Figure 6-4 Selecting the dashboard

2. On the dashboard, navigate to the Services section and click the IBM Cloud Object Storage instance that you created earlier, as shown in Figure 6-5.



Services

Name	Location	Resource Group	Plan	Details	Service Offering
COS-learning	global	default	Lite	Provisioned	Cloud Object Stora
Internet Services- COS-learning	global	default	Early Access	Provisioned	Internet Services
Key Protect-COS	United Kingdom	default	Graduated Tier Pric	Provisioned	Key Protect
rc9d57eb53-a49e-4956-8adb-1bcaaeb79cd6	US South	default	Graduated tier	Provisioned	App ID

Figure 6-5 Services section of the dashboard

3. Explore the IBM Cloud Object Storage web console. The navigation on the left lets you explore Buckets, Endpoints, Service Credentials, and so on, as shown in Figure 6-6.

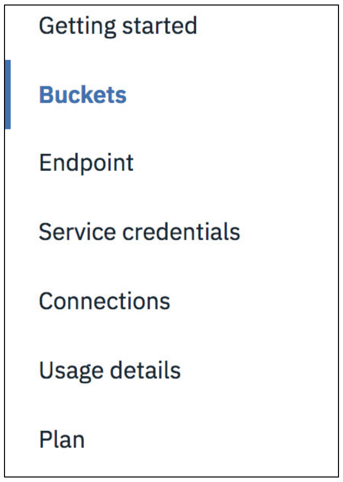


Figure 6-6 Left navigation pane

The Navigation pane provides these options:

– Buckets, as already mentioned, are a logical unit of storage used to group objects.

– Endpoints provide a list of the API endpoints that are available to use. IBM Cloud Object Storage offers cross-region, regional, and single site endpoints. Which one you select affects where your data is stored as well as the resiliency and availability. For example, cross-regional has the best resiliency and availability, but potentially higher latency because your data is dispersed across multiple sites in a wider geographic location.

– Service Credentials is where you can generate API or HMAC keys to authenticate with IBM Cloud Object Storage. You can use these keys in your application or a compatible tool.

4. Now you are ready to start writing some code. First, install the modules that you will need from the `package.json` file:

```
$ npm install
```

For clarity, the following Node.js modules are being used:

– `ibm-cos-sdk`: The IBM Cloud Object Storage SDK for Node.js

– `ibm-cloud-env`: The IBM Cloud Env, which abstracts the underlying runtime environment so that apps are portable between your notebook, Cloud Foundry, and Kubernetes

– `log4js`: The logging framework for Node.js

– `uuid`: Generates universal unique identifiers

– `multer`: Express.js plug-in for handling multipart form data uploads

– `imagemagick`: Node.js wrapper for the image manipulation utility

– `which`: Find the first instance of an executable file in the PATH variable

– `dms2dec`: Degrees, minutes, seconds (sexagesimal) for decimal GPS positions, which is useful for parsing PGS exif tags in geotagged images

5. Create a folder in `server/services` called `cos` and create a new file in that directory called `index.js`:

   ```
   $ mkdir -p server/services/cos
   $ touch server/services/cos/index.js
   ```

Generally, split code into modules so that it is easy to reuse and to separate concerns. Throughout this module series, you create new services and housing them in their own file to divide the behavior of your application into reusable modules.

Example 6-1 shows the code in `server/services/cos/index.js` that reads and writes objects to IBM Cloud Object Storage using the SDK. You will integrate this code into your existing application logic so a user can upload a file and it will be stored in IBM Cloud Object Storage.

Example 6-1 Code from server/services/cos/index.js

```
"use strict";

// Modules
const COS = require("ibm-cos-sdk"); // the COS SDK
const IBMCloudEnv = require("ibm-cloud-env"); // access cloud environment vars
const log4js = require("log4js");

// Bucket names - taken from environment variables set in Helm chart
const _originalBucket = IBMCloudEnv.getString("cos_original_bucket");
const _normalizedBucket = IBMCloudEnv.getString("cos_normalized_bucket");

// global logger object
const logger = log4js.getLogger("cos-service");


// Config - Service credentials for the COS instance
const config = {
   endpoint: IBMCloudEnv.getString("cos_endpoint"),
   apiKeyId: IBMCloudEnv.getString("cos_api_key"),
   ibmAuthEndpoint: IBMCloudEnv.getString("cos_ibm_auth_endpoint"),
   serviceInstanceId: IBMCloudEnv.getString("cos_service_instance_id"),
};
const cos = new COS.S3(config);

// Public methods
module.exports.doCreateObject = _doCreateObject;
module.exports.doReadObject = _doReadObject;
module.exports.generatePresignedGet = _generatePresignedGet;
module.exports.originalBucket = _originalBucket;
module.exports.normalizedBucket = _normalizedBucket;

// Private methods

function _doCreateObject(bucketName, mimeType, key, data) {
   let params = {
      Bucket: bucketName,
      // ContentEncoding: 'base64', - optional
      ContentType: mimeType,
      Key: key,
      Body: data
   };
```

```
   return new Promise(function(resolve, reject) {
      cos.putObject(params, function(err, data) {
         if(err) {
            reject(err);
         }
         else {
            logger.info(data);
            resolve(data);
         }
      });
   });
}

function _doReadObject(key) {
   logger.info("Getting object with key", key);
   return new Promise(function(resolve, reject) {
      cos.getObject({
         Bucket: _normalizedBucket,
         Key: key
      }, function(err, data) {
         if(err) reject(err);
         resolve(data);
      });
   });
}

// Given a key, generate a pre-signed URL that can be used to GET an image.
// More info on pre-signed URLs here:
function _generatePresignedGet(key) {
   logger.info("Generating presigned GET URL for key", key);
   return new Promise(function(resolve, reject){
      var url = cos.getSignedUrl("getObject", {
         Bucket: _normalizedBucket,
         Key: key
      });
      if (url!= null) {
         resolve(url);
      } else {
         reject("Failed to generate pre-signed URL");
      }
   });
}
```

Copy the code in Example 6-1 on page 69 into the file that you created, `server/services/cos/index.js`.

To begin, load the modules that you need for this service using a `require` statement. Then, configure the service by getting the credentials and bucket names from the environment where they have been injected using the IBM Cloud env package. For more information about how credential injection works, see Chapter 4, “Scaffolding using a Starter Kit” on page 29.

Throughout these modules, we are following a pattern where you define private methods, such as `_doCreateObject` that are preceded with an underscore. Then explicitly expose the methods that you want other parts of your app to consume by adding them to the `module.exports` object.

The IBM Cloud Object Storage service consists of two functions: `doCreateObject` and `doReadObject`. The names are fairly self-explanatory as to what they do, and both return `Promises`. This application takes advantage of `Promises` when performing asynchronous actions. These are actions that do not immediately return a result, such as requesting data from an API or reading a file from the file system. For more information about `Promises`, see "Promise" at MDN web docs.

`_doCreateObject` expects these parameters:

- `bucketName` is the name of the bucket that the object will be stored in. This function assumes the name of the bucket that it is given already exists.
- `mimetype` is a two-part identifier for file formats commonly used in HTTP transfers. It is important to include this parameter so when the object is downloaded again, the system knows what kind of file it is.
- `key` is the name of the object as it appears in IBM Cloud Object Storage. The key must be unique within the bucket and is used when you want to read, delete, or update that object from the bucket. In this application, the key is in the following format:

  ```
  original | normalized | claim ID | timestamp | filename
  ```

  There are several reasons that we decided on this naming convention:

  – Indicates the quality of the photo
  – Easily readable by humans and machines
  – Guaranteed to be unique
  – Organized hierarchically such that SQL Query will be able to process it efficiently (see Chapter 5, "Storing metadata in Cloudant" on page 51)
- `data` is either a `Buffer`, `Typed Array`, `Blob`, `String`, or a `ReadableStream`.

`_doReadObject` expects a single parameter, `key`. `key` is the object name assigned to it when it was created. We already know which bucket we want to get the image from because we only want to work with the normalized images.

Now integrate the IBM Cloud Object Storage service into your main application flow. You must create an API route, `/:claimId/image`, that the front end of this application will use to submit images uploaded by the user. Then, you will store those images on IBM Cloud Object Storage. Create a folder called `claims` in `server/routers` and create a file called `postClaimImage.js`:

```
$ mkdir -p server/routers/claims
$ touch server/routers/claims/postClaimImage.js
```

Copy the code in Example 6-2 and paste it into `postClaimImage.js`. This code snippet imports the node modules that you need and sets up an Express.js route, `/:claimId/image`, that uses Multer middleware to accept `multipart/formdata` uploads and assign the function `_processImage` to handle that route. The `_setupRoute` function is exported as a variable called `init`.

Example 6-2 Code to import node modules

```
const util = require("util");
const fs = require("fs");
const uuid = require("uuid/v1");
const cos = require("../../services/cos/index");
const imageProcessor = require("../../services/image-processing/processor");
const __ = require("lodash");
const multer = require("multer");
```

```
const log4js = require("log4js");

const logger = log4js.getLogger("claim-post-image");

function _setupRoute(router) {
   return router.post("/:claimId/image", upload.any(), _processImage);
}

module.exports.init = _setupRoute;
```

Copy the code in Example 6-3 and paste it into `postClaimImage.js`. Now configure the Multer middleware. Multer can be used to process any kind of multipart form data, but we are only interested in accepting jpg or png images. Therefore, create a file filter that checks the file extension and accepts or rejects the file. Also, configure Multer to use disk storage and store uploaded images in a temporary directory.

Example 6-3 Code for processing image files

```
/*
 * Filters which files multer should accept to be uploaded
 * Only accept .jpg or .png files
 */
function multerFileFilter(req, fileDetails, cb) {
   let file = fileDetails.originalname;
   let fileName = file
      .substring(file.lastIndexOf("/") + 1, file.lastIndexOf("."));
   let extension = file
      .substring(file.lastIndexOf("."), file.length).toLowerCase();
   logger.info(fileName, extension);
   if (extension == ".jpg" || extension == ".png" || extension == ".jpeg") {
      // accept the file
      cb(null, true);
   } else {
      // reject the file, not a jpg or png
      cb(null, false);
   }
}

// configure multer - multipart upload for form-data
const storage = multer.diskStorage({
   destination: function (req, file, cb) {
      // create the dir to store the images on disk
      var dir = "/tmp/image-uploads";
      if (!fs.existsSync(dir)){
         fs.mkdirSync(dir);
      }
      cb(null, dir);
   },
   filename: function (req, file, cb) {
      cb(null, uuid() + "-" + file.originalname);
   }
});
```

```
const upload = multer({
   storage: storage,
   fileFilter: multerFileFilter
```

Copy the code in Example 6-4 and paste it into `postClaimImage.js`. The process image function handles the image upload POST request. Create two variables to store data about the image and the claim, `image` and `append`, to store in Cloudant.

Example 6-4   Code for image upload POSt request

```
/*
 * 1. Receive image
 * 2. Extract metadata
 * 3. Resize image
 * 4. Send image to VR service
 * 5. Store images in COS
 * 6. Reassemble JSON from various API calls and store in Cloudant
 */
function _processImage(req,res) {

   // create a unique identifier to use as the key of the images object
   // so we can use it as a reference
   const imageId = uuid();
   let image = {};
   let imageContainer = {};
   imageContainer[imageId] = image;

   const append = {
      claimId: req.params.claimId,
      images: [imageContainer]
   };

   // create a timestamp to use when uploading the images
   let dateTime = Date.now();

   let file = req.files[0];

   // read the uploaded image file from the tmp directory
   new Promise((resolve, reject) => {
      fs.readFile(file.path, (err, data) => {
         if (err) {
            reject(err);
         } else {
            resolve(data);
         }
      });
   })
      .then(data => {
         // create a unique key for object storage
         let origKey = 
`original/${req.params.claimId}/${dateTime}/${file.filename}`;
         // store key of original image in claimImageRecord for later retrieval
         image.original = origKey;

         // store the original user-uploaded image in a COS bucket
         return cos.doCreateObject(cos.originalBucket, file.mimetype, origKey, 
data);
      })
      .then(() => {
         logger.info("Uploaded original image to COS");

         // resize the image to get an normalized thumbnail
         return imageProcessor.resize(file.path);
      })
      .then(resizedImg => {
         // store the normalized image in a COS bucket
         let normKey = 
`normalized/${req.params.claimId}/${dateTime}/${file.filename}`;
         // store key of normalized image in claimImageRecord for later retrieval
         image.normalized = normKey;

         return cos.doCreateObject(cos.normalizedBucket, file.mimetype, normKey, 
resizedImg);
      })
      .then(function() {
         // append all the data in claimImageRecord to Cloudant
         return req.cloudant.appendClaim(append);
      }).then(() => {
         res.status(200).end();
      })
      .then(() => {
         fs.unlinkSync(file.path);
      })
      .catch((err) => {
         logger.debug("in the CATCH block", err);
         logger.debug(util.inspect(err));
         fs.unlinkSync(file.path);
         res.status(500).send(err.message);
      });
```

Then, read the file that Multer stored on the disk and get the image data as a buffer (the IBM Cloud Object Storage module expects a buffer). Upload the original image to be stored in IBM Cloud Object Storage. Next, use `processor.js` to resize the image to a normalized format and then store the normalized image in IBM Cloud Object Storage. Finally, delete the original image from the disk. After this process is completed using a Promise chain, append the keys for images in IBM Cloud Object Storage to the claim that the customer is making. That way, when the front end requests the images as thumbnails to display as part of the claim, it can use the keys that are stored in the claim document.

By returning promises in your IBM Cloud Object Storage service, you can chain a series of promises. This technique makes it easy for you to add functions to the app later, such as weather data extraction or visual recognition classification.

The HTML in the front end to handle the image upload is shown below. It is a basic use of a form and some JavaScript code.

```
<form>
    <input type="file" name="userfile" id="userfile" />
    <br/>
    <input type="button" id="upload" value="upload" />
</form>
```

The JavaScript that uploads the image is shown in Example 6-5.

Example 6-5 Uploading images JavaScript

```
$('#upload').click(function(){
   console.log('upload button clicked!')
   var fd = new FormData();
   fd.append( 'userfile', $('#userfile')[0].files[0]);
   var claimId =  document.getElementById("claimSummaryId").innerHTML;
   console.log(claimId);
   var postImageURL = "/claim/" + claimId + "/image";
   var uploadInProgessMessage = "Upload In Progress...";
   document.getElementById("uploadProgress").innerHTML = uploadInProgessMessage;
   $.ajax({
     url: postImageURL,
     data: fd,
     processData: false,
     contentType: false,
     type: 'POST',
     success: function(data){
       console.log('upload success!')
       var uploadSuccessMessage = "Upload Succeeded";
       document.getElementById("uploadProgress").innerHTML = uploadSuccessMessage;
       $('#data').empty();
       $('#data').append(data);
       retrieveClaimDetails(claimId);
     },
     error: function(error) {
       var uploadFailedMessage;
       if(error) {
         uploadFailedMessage = error.statusText + " - " + error.responseText
       } else {
         uploadFailedMessage = "Upload Failed";
       }
       document.getElementById("uploadProgress").innerHTML = uploadFailedMessage;
     }
   });
});
```

The code in Example 6-6 returns an image from IBM Cloud Object Storage when supplied with its key. Create a file in `server/routers/claims` called `getClaimImage.js`:

```
$ touch server/routers/claims/getClaimImage.js
```

Example 6-6 Code to return image from IBM Cloud Object Storage

```
const cos = require("../../services/cos/index");

module.exports.init = _setupRoute;

function _setupRoute(router) {
   return router.get("/:claimId/image", _getImage);
}

function _getImage(req, res) {
   // get the key of the image from the query parameters
   let key = req.query.key;
   // check to make sure we have a key
   if(key) {
      // read the object from COS
      cos.doReadObject(key).then(image => {
         // set some headers for type, length and disposition
         res.writeHead(200, {
            "Content-Type": image.ContentType,
            "Content-disposition": "attachment;filename=" + key,
            "Content-Length": image.ContentLength
         });
         // return the image to the user
         res.end(image.Body);
      });
   } else {
      // send an error, we don't have a key
      res.status(500).send("Please include query parameter 'key' " +
      "to specify the image key");
   }
}
```

Import the necessary modules, export an `init` function, and create an API route, `/:claimId/image`. This route allows the front end to do a GET request specifying the `claimId` and the key of the image in IBM Cloud Object Storage, and your API will return that image. The `_getImage` function gets the key from the query string of the request, and uses the IBM Cloud Object Storage module to read an object from IBM Cloud Object Storage. Because it is the front end making the request, you want the image to come from the normalized bucket.

You have now created routes for uploading an image and getting an uploaded image. You must register these routes in `server/routers/claim/index.js` so they become part of the `/claim` route, as shown in Example 6-7.

Example 6-7 Registering the routes

```
const express = require("express");
const passport = require("passport");
const serviceManager = require("../../services/service-manager");

const log4js = require("log4js");
const common = require("../common");
const postClaimImage = require("./postClaimImage").init; // ADD THIS LINE
const getClaimImage = require("./getClaimImage").init; // ADD THIS LINE

const bodyParser = require("body-parser");
const logger = log4js.getLogger("claim-controller");
```

Require the two new modules that you have created, `postClaimImage` and `getClaimImage`, so that your code in `server/routers/claim/index.js` looks like the snippet above (Example 6-8).

Example 6-8 Requiring the modules

```
module.exports = function(app) {
   const router = express.Router();

   postClaimImage(router); // ADD THIS LINE
   getClaimImage(router); // ADD THIS LINE
   // Ensures claim exist and attaches it to the request
   const claimInjector = function (req, res, next) {
      // get the claimId from the request parameters
      const claimId = req.params.claimId;

      req.cloudant.getClaim(claimId).then((claimDetails) => {
         if (claimDetails) {
            req.claim = claimDetails;
            next();
         } else {
            res.status(404).send("Claim does not exist");
         }
      }).catch((err) => {
         logger.error(err);
         res.status(500).send("Something broke!");
      });
   };
```

Call the `init` function for both modules and pass in the Express router to initialize the modules. Make sure that your code in `server/routers/claim/index.js` looks like the snippet above. So now you have created the modules to handle upload and download of images and added them to your `/claim` route.

`postClaimImage.js` uploads images. However, you still must create the module that will resize the uploaded images. Create a file called `processor.js` in `server/services/image-processing`:

```
$ mkdir -p server/services/image-processing
$ touch server/services/image-processing/processor.js
```

Paste the code in Example 6-9 into `server/services/image-processing/processor.js`. The code snippet is responsible for resizing the image. A utility called ImageMagick is used to do this task, which you must install as part of your Docker build. Then, use a small Node.js wrapper library around Image Magick.

Example 6-9   Code to resize the image

```
"use strict";

const im = require("imagemagick");
const which = require("which");
const dms2dec = require("dms2dec");
const log4js = require("log4js");

// global logger object
const logger = log4js.getLogger("image-service");

// Public methods

// guard function to ensure imageMagick is there before we call the im package
const _checkImageMagick = (func, ...args) => {
   // check if preconditions are met (imagemagick is installed)
   if (!which.sync("convert", {nothrow: true})) {
      throw new Error("ImageMagick does not appear to be installed, please install
it!");
   }
   return func(...args);
};

module.exports.resize = (...args) => _checkImageMagick(_resize,...args);

// Private methods

function _resize(imageData) {
   return isImagePortrait(imageData).then(isPortrait => {
      // define our width and heigh to resize to
      let width = 1024;
      let height = 768;
      // if the image is portrait, swap values of width and height so we can
      // maintain the ratio
      if(isPortrait) {
         logger.info("Resize: swapping width & height values to maintain ratio");
         width = 768;
         height = 1024;
      }
      return new Promise(function(resolve, reject) {
         im.resize({
            srcPath: imageData,
            width: width,
            height: height
         }, function(err, stdout) {
            if (err) {
               reject(err);
            } else {
               logger.info("resized image to fit within 768px");
               // Convert the string data from the resize function to a binary
buffer
               var buff = Buffer.from(stdout, "binary");
               resolve(buff);
            }
         });
      });
   });
}

function getImageDimensions(imageData) {
   return new Promise(function(resolve) {
      im.identify(["-format", "%wx%h", imageData], function(err, output){
         if (err) resolve(err);
         let dimensions = output.split("x");
         resolve(dimensions);
      });
   });
}

function isImagePortrait(imageData) {
   return getImageDimensions(imageData).then(dimensions => {
      let ratio = dimensions[0] / dimensions[1];
```

```
        if(ratio < 1) return true;
        return false;
    });
```

Insurance Company X would like to eventually archive inactive claims after a certain time. This is a good practice because otherwise the main database will grow to be a very large size that costs more money and affects performance when indexing. You can archive the JSON data directly into IBM Cloud Object Storage. A service on IBM Cloud called SQL Query allows you to query files directly from IBM Cloud Object Storage, including JSON, CSV, and Parquet documents. You can read the documentation for more information.

Replace the code for the `_archiveClaim` function in `server/services/cloudant/index.js` with the code snippet in Example 6-10. Notice the line that uses the IBM Cloud Object Storage module to create an object in a bucket called `claims-archived`.

Example 6-10 Code for archiving

```
function _archiveClaim(claimId) {
    // get the claim, store it on COS and update the claim as archived
    return _getClaim(claimId)
        .then((claimDetails) => {
            const data = JSON.stringify(claimDetails);
            logger.trace("Will archive: " + data);
            // prefix object by year, week and claimId, such that hive partitioning
may benefit from it
            const now = moment.utc();
            return cos.doCreateObject("claims-archived",
`${now.year()}/${now.week()}/${claimId}/data.json`, "application/json", data);
        })
        .then(() => _appendClaim({ claimId: claimId , state: "archived"}));
}
```

6.4.2 Testing the app

For this app to run locally on your computer, you need to install the ImageMagick utility. For Mac users, you can install Homebrew and issue the following command:

```
$ brew install imagemagick
```

For other operating systems, there are download instructions.

To test the app, complete these steps:

1. Issue the following command:

   ```
   $ npm start
   ```

2. Open your browser and navigate to `http://localhost:3000`.
3. Click **Login** in the upper right of the window.
4. Click the blue **Make a Claim** button.
5. Using the form, choose an image to upload (pick an image from the `testData` directory). You can see in the terminal window the application logging the steps that it takes to upload an image. After the upload is complete, the image is displayed in the window.

6.4.3 The IBM Cloud Object Storage SDK and singlepart versus multipart uploads

The IBM Cloud Object Storage Node.js SDK can perform many more actions for you and supports the entire IBM Cloud Object Storage API command set. The full documentation can be found here. The following are examples of other tasks you can perform with the SDK:

- Create, read, update, and delete buckets
- Create, read, update, and delete objects
- Enable CORS on a bucket
- Creating a multipart upload

The final item in the list, creating a multipart upload, is worth discussing. IBM Cloud Object Storage supports two methods of uploading files: Single-part and multipart. In this example, we are using `singlepart` because we are uploading a single image at a time over a web browser over a reliable internet connection.

In a `singlepart` upload, the entire object is uploaded as a single part. If the upload fails, the process must start from the beginning again, and there is little indication of the progress of the upload. Contrast this upload with multipart, where an object or multiple objects are divided into parts and a separate upload is performed for each part. This technique can be achieved by using the IBM Cloud Object Storage API, but the SDK handles the complexities of creating the parts and managing the entity tags and part numbers.

The following are some example scenarios that benefit from multipart uploads:

- Uploading large files
- On mobile devices with unreliable network connections, dividing uploads into parts reduces the amount that must be retransmitted in the event of an upload failure

`Multipart` uploads consist of three steps: Initiate the upload, upload the object parts, and then complete the `multipart` upload. When IBM Cloud Object Storage receives the multipart upload completed request, the parts are constructed into a single object that you can then access like any other object.

`Multipart` uploads are only available for objects larger than 5 MB. For objects smaller than 50 GB, a part size of 20 MB to 100 MB provides optimum performance. For larger objects, part size can be increased without significant effect on performance. Using more than 500 parts leads to inefficiencies in object storage, and should be avoided when possible.

Sometimes a multipart upload will fail. To avoid storing incomplete objects and therefore incurring unnecessary costs, call `abortMultipartUpload`. To verify whether a multipart upload has failed, first call `listParts`. If this list is not empty, then the multipart upload has failed and you should abort it.

Due to the additional complexity involved, use S3 API libraries that provide `multipart` upload support. For more information, see 6.4.4, "Using ManagedUpload or TransferManager".

6.4.4 Using ManagedUpload or TransferManager

`ManagedUpload` provides a simple API for uploading content to IBM Cloud Object Storage and makes extensive use of multipart uploads to achieve enhanced throughput, performance, and reliability.

When possible, `ManagedUpload` attempts to upload multiple parts of a single upload at once. When dealing with large content sizes and high bandwidth, this process can significantly increase throughput.

`ManagedUpload` handles a lot of complexity for you, including dividing objects into multiple parts, aborting failed uploads, returning progress events, adjusting the algorithm for maximum performance, and presenting a simple API for ease of use.

The Managed uploader has an equivalent in the Java SDK known as the `TransferManager`. Documentation can be found here.

6.5 Next steps

Review what you have learned and move on to Chapter 7, “Advanced Object Storage integration patterns” on page 83.

6.6 Other references

- MIME types (a complete list)

  `https://www.sitepoint.com\mime-types-complete-list`
- IBM Cloud Object Storage Node.js SDK documentation

  `https://ibm.github.io/ibm-cos-sdk-js/`
- IBM Cloud Object Storage Java SDK documentation

  `https://ibm.github.io/ibm-cos-sdk-java`
- IBM Cloud Object Storage documentation

  `https://console.bluemix.net/docs/services/cloud-object-storage/`

7

Advanced Object Storage integration patterns

Welcome to Module 7 in this tutorial series! This series explains how to build a cloud-native, consumer-facing car insurance claim application on IBM Cloud that is built according to best practices, uses IBM Cloud Object Storage at its core, and employs AI and machine learning to provide a unique digital experience for a customer.

This module explains some of the advanced techniques available to a developer when interacting with Cloud Object Storage, and describes the use cases in which each of these techniques can add value to an application.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

7.1 Learning objectives

After completing this module, you will be able to:

- Identify the best advanced method to interact with Cloud Object Storage for your use case
- Implement code to write data to Cloud Object Storage using a single part
- Implement code to write data to Cloud Object Storage using multipart upload and concurrent streams
- Implement code to read specific data ranges from Cloud Object Storage objects
- Verify the integrity of data stored in and retrieved from Cloud Object Storage
- Implement code to create pre-signed URLs for others to interact with Cloud Object Storage objects
- Identify the tiering options provided by IBM Cloud Object Storage and choose the tier that best suits your data

7.2 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 6 to get the finished code from the previous module that you will build on:

```
$ cd module06
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module07
```

As with previous modules, this section assumes that you have an active IBM Cloud account with Object Storage provisioned. This section also assumes that you have the IBM Cloud Object Storage SDK for Node.js installed. If you do not, open a terminal and enter the following command to install it:

```
npm install ibm-cos-sdk
```

7.3 Implementation

As discussed in previous modules, Cloud Object Storage provides a number of methods for accessing and interacting with the service. This module will focus on some of the advanced techniques afforded by the IBM Cloud Object Storage SDKs that are not explicitly used in the example application.

7.3.1 Uploading data to Cloud Object Storage

When sending data to Cloud Object Storage, the operator should choose the most efficient method for transferring the data: As a single part, or broken up into multiple, smaller chunks of data (multipart upload).

Each method has its own set of advantages and disadvantages.

Table 7-1 shows the pros and cons of single part transfers.

Table 7-1 Advantages and disadvantages of single part transfer

Pro	Con
Simple Implementation Call `cos.putObject` and wait for the callback	Upload throughput bounded by single part
Easy data integrity checking ETag will equal the md5sum of the source data	Cannot monitor upload progress
Relatively easy to debug Any errors will surface in the method's callback	Transmission failure requires resending the entire object

Table 7-2 shows the pros and cons of multipart upload transfers.

Table 7-2 Advantages and disadvantages of multipart upload transfers

Pro	Con
Increased upload throughput with concurrency	Data integrity checking is more involved
Transmission failure requires only resending the failed part	Debugging can be more difficult because each part upload has its own error callback
Easy to monitor status	Requires cleanup of uploaded data on abort
	Manual multipart upload requires several steps

Although there are no fixed criteria for making the determination between single part and multipart upload, two major factors can help you determine which method to employ: The size of the data to be uploaded, and the available bandwidth and network reliability of the client that will upload the data.

As the size of the data to be uploaded and the bandwidth available to the client increase, moving from a single part transfer to a concurrent operation can result in a significant increase in overall throughput.

Multipart uploads are only available for objects larger than 5 MB. For objects smaller than 50 GB, use a part size of 20 MB to 100 MB for optimum performance. For larger objects, part size can be increased without significantly affecting performance.

Using more than 500 parts leads to inefficiencies in Object Storage and should be avoided when possible.

Standard upload (single part)

Begin with a standard upload operation. As in previous modules, you will split the code that interacts with the Cloud Object Storage service into a service module (`cos_advanced.js`) and require it in your example code.

Create a file named `cos_advanced.js` now and enter the code shown in Example 7-1, replacing the placeholder values with those specific to your IBM Cloud Object Storage account.

Example 7-1 Service module code

```
"use strict";

// Modules
const COS = require('ibm-cos-sdk'); // the COS SDK
const util = require('util'); // used to inspect errors
const fs = require('fs'); // used to read/write from the file system

// COS bucket name
const _bucketName = 'example_bucket'; // replace this with your bucket name

// Config - Service credentials for the COS instance
const config = {
  endpoint: 's3.us-south.objectstorage.softlayer.net', // replace this with your
endpoint if deployed to a different region
  accessKeyId: 'my_access_key_id', // replace this with your access key id
  secretAccessKey: 'my_secret_access_key_id' // replace this with your secret
access key
};
const cos = new COS.S3(config);

// Public methods
module.exports.doSinglePartUpload = _doSinglePartUpload;

// Private methods
// upload an object with a single part
function _doSinglePartUpload(mimeType, key, data) {
  console.log('Uploading object with key', key);
  let params = {
    Bucket: _bucketName,
    ContentType: mimeType,
    Key: key,
    Body: data
  };
  return new Promise(function(resolve, reject) {
    cos.putObject(params, function(err, data) {
      if(err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}
```

The `doSinglePartUpload` function accepts a MIME Type (ex. `'text/plain'`, `'application/octet-stream'`), a key (the `'name'` of the object to be uploaded), and the data to be uploaded to the given key. This function returns a Promise that can be resolved later to determine whether the operation was successful.

Create a file named `simple_put.js` in the same directory as `cos_advanced.js` and enter the code shown in Example 7-2.

Example 7-2 Code for simple_put.js

```
const cos = require('./cos_advanced');

// Simple single part PUT object
cos.doSinglePartUpload('text/plain', 'example1.txt', 'hello,
world!').then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error);
})
```

Our first foray into uploading to Cloud Object Storage is a simple one. Upload a `text/plain` (`mimeType`) object to `example1.txt (key)` with the most traditional of contents: '`hello, world!' (data)`. Then send the result (`res`) in case of success, or the error (`error`) in case of failure to the console for inspection.

Next, run `simple_put.js`:

```
~# node simple_put.js
Uploading object with key example1.txt
{ ETag: '"3adbbad1791fbae3ec908894c4963870"' }
```

If you see the `ETag` returned from the object storage system, congratulations, the object is uploaded. As with all writes to IBM Cloud Object Storage, after the `PUT` is acknowledged, the uploaded data is immediately available to read. This code suffices for uploading simple strings to Cloud Object Storage, but not for anything more complicated.

Create a file named `simple_put_file.js` in the same directory and enter the code shown in Example 7-3.

Example 7-3 Code for the simple_put_file.js file

```
const cos = require('./cos_service');
const fs = require('fs');

let buffer = fs.readFileSync('hello_world.txt');

// Simple single stream PUT of file contents
cos.doSinglePartUpload('text/plain', 'example2.txt', buffer).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error);
})
```

The `doSinglePartUpload` function accepts a `string` or a `buffer` as its data parameter, meaning you can also feed it the contents of a file on disk and upload it to the object storage system. This `simple_put_file.js` example reads the contents of a file named `hello_world.txt` into a buffer and passes that buffer as the data to the `doSinglePartUpload` function.

Next, run the following commands:

```
~# echo 'hello, world!' > hello_world.txt
~# node simple_put_file.js
Uploading object with key example2.txt
{ ETag: '"910c8bc73110b0cd1bc5d2bcae782511"' }
```

This procedure creates the `hello_world.txt` file that your `simple_put_file.js` script will read from, and then run the code. Just as before, it returns successfully, and an ETag value is printed to console. The implementation above is sufficient for many use cases. Because the components in this tutorial series are deployed into the same datacenters as the Cloud Object Storage system and the objects created by these components are relatively small, this technique is used for uploading data to Cloud Object Storage throughout this series.

Multipart upload (manual)

In some cases, especially as the data to be uploaded grows in size (e.g. over 100 MiB), uploading the data in pieces provides several advantages over a single, 'atomic' operation. These benefits include optional concurrency (the operator can elect to initiate the upload of parts while others are already underway), and more granular control of errors and retry logic (the operator can retry a failed part instead of the entire object in the event of error).

Multipart uploads occur in three main phases, as shown in Table 7-3.

Table 7-3 Phases of multipart uploads

Phase	Function
Begin the upload	`createMultiPartUpload`
Upload part(s)	`uploadPart`
End the upload	`completeMultiPartUpload` / `abortMultiPartUpload`

First, the operator creates the upload with a call to `createMultiPartUpload`. If this is successful, `uploadPart` can be called as many times as necessary to transfer all of the data to Cloud Object Storage. If all parts are uploaded successfully, the operator can then call `completeMultiPartUpload` to finalize the upload to clean up the parts storage and finalize the object.

If an error occurs, the operator can either attempt to re-upload the failed part, or call `abortMultiPartUpload` to cancel the upload and clean up the parts storage to ensure that you are not charged for storage of the partially uploaded object.

Open `cos_advanced.js` and update the “public methods” and “Private methods” sections to include the code shown in Example 7-4.

Example 7-4 Updating the cos_advanced.js code

```
// Public methods
module.exports.doSinglePartUpload = _doSinglePartUpload;
module.exports.doCreateMultiPartUpload = _doCreateMultiPartUpload;
module.exports.doAbortMultiPartUpload = _doAbortMultiPartUpload;
module.exports.doCompleteMultiPartUpload = _doCompleteMultiPartUpload;
module.exports.doUploadPart = _doUploadPart;

// Private methods

// create a new multipart upload
```

```
function _doCreateMultiPartUpload(key) {
  console.log('Creating multipart upload with key', key);
  let params = {
    Bucket: _bucketName,
    Key: key
  };
  return new Promise(function(resolve, reject) {
    cos.createMultipartUpload(params, function(err, data) {
      if(err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}

// upload a part in a multipart upload operation
function _doUploadPart(key, partNumber, uploadId, data) {
  console.log('Uploading part ' + partNumber + ' for key ' + key + ' with uploadId
' + uploadId);
  let params = {
    Bucket: _bucketName,
    Key: key,
    PartNumber: partNumber,
    UploadId: uploadId,
    Body: data
  };
  return new Promise(function(resolve, reject) {
    cos.uploadPart(params, function(err, data) {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}

// abort an in-progress multipart upload operation
function _doAbortMultiPartUpload(key, uploadId) {
  console.log('Aborting multipart upload for key ' + key + ' with uploadId ' +
uploadId);
  let params = {
    Bucket: _bucketName,
    Key: key,
    UploadId: uploadId
  };
  return new Promise(function(resolve, reject) {
    cos.abortMultipartUpload(params, function(err, data) {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}

// complete an in-progress multipart upload operation
function _doCompleteMultiPartUpload(key, uploadId) {
  console.log('Completing multipart upload for key ' + key + 'with uploadId ' + 
uploadId);
  let params = {
    Bucket: _bucketName,
    Key: key,
    UploadId: uploadId
  };
  return new Promise(function(resolve, reject) {
    cos.completeMultipartUpload(paramse, function(err, data) {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
}
```

Each function above corresponds to a phase of the multipart upload operation and wraps the step in a promise for ease of use outside of the service module.

Next, create a file named `multipart_create_and_abort.js` and add the code in Example 7-5.

Example 7-5 Code for multipart_create_and_abort.js

```
const cos = require('./cos_advanced');
let key = 'multipart.dat'; // the object name

// create a multipart upload, then immediately abort
cos.doCreateMultiPartUpload(key).then(function(res) {
  console.log(res);
  cos.doAbortMultiPartUpload(key, res.UploadId).then(function(res) {
    console.log(res)
  });
}).catch(function(error) {
  console.error(error)
});
```

This code creates a multipart upload. When a multipart upload is created successfully, the Cloud Object Storage SDK returns a JavaScript object that includes the bucket name, the object's key, and, importantly, the `UploadId`. It is essential when manually managing multipart uploads to keep the `UploadId` close to hand. Any follow-up actions in the multipart upload flow require this ID as a parameter.

Because the create operation returns a promise, you can trigger a following action with `.then` to abort the multipart upload. Calls to `doAbortMultiPartUpload` require both the key and the `UploadId` of the transaction to be cancelled. Because you specified the key yourself, this is easy to provide. `doCreateMultiPartUpload` returns the response when its promise is successfully resolved, so you can access the `UploadId` as a property of the response (`res`).

Creating and immediately aborting an upload has little real-world value, but will let us see the results of each operation as we build up to a complete multipart upload operation.

Run `multipart_create_and_abort.js`, as shown in Example 7-6.

Example 7-6 multipart_create_and_abort.js command

```
~# node multipart_create_and_abort.js
Creating multipart upload with key multipart.dat
{ Bucket: 'examplebucket',
  Key: 'multipart.dat',
  UploadId: '01000163-4307-b006-1e57-151586b0057e' }
Aborting multipart upload for key multipart1.dat with uploadId
01000163-4307-b006-1e57-151586b0057e
{}
```

When you run this code, you see the debug message from the `cos_advanced.js` module acknowledging that it is creating a multipart upload. When that multipart upload creation succeeds, the code logs the response and see the bucket name, object key, and `UploadId` returned. Next, the returned `UploadId` and your original `key` are passed to the following function to cancel the multipart upload. When the abort returns, you receive an empty object.

Now that creating and aborting multipart uploads are well in hand, you can move on to a complete multipart upload.

Create a file named `multipart_create_upload_and_complete.js` and add the code in Example 7-7.

Example 7-7 Code for multipart_create_upload_and_complete.js

```
const cos = require('./cos_advanced');
let key = 'multipart.dat'; // the object name
let buffer = new Buffer(1024 * 1024 * 25); // a 25MB buffer

// create a multipart upload, upload 4 25MB parts, then complete
cos.doCreateMultiPartUpload(key).then(function(res) {
  console.log(res);
  var put_promises = new Array();
  for (var i=0;i<4;i++) {
    put_promises.push(cos.doUploadPart(key, i + 1, res.UploadId, buffer));
  }
  Promise.all(put_promises).then(function(parts) {
    for (var j=0; j<parts.length;j++) {
      parts[j].PartNumber = j+1;
    }
    cos.doCompleteMultiPartUpload(key, res.UploadId, parts).then(function(res) {
      console.log(res)
    });
  })
}).catch(function(error) {
  console.error(error)
});
```

This script has a bit more going on than the previous examples. At the top of the file, you need your Cloud Object Storage service module, then create a key variable to hold the name of your object. Line 3 creates a new empty buffer 25 MB in size.

As before, create a new multipart upload and pass it in the key. Next, declare an array prototype named `put_promises` in which you will collect the promises representing your upload operations.

The first for loop increments from 0 to 3, creating promises and adding them as four distinct calls to `doUploadPart` to the `put_promises` array.

Note: In the call to `cos.doUploadPart`, we are adding 1 to our loop counter (i + 1) before passing it as the `partNumber` parameter. We do this because `partNumber` is 1-indexed. `doUploadPart` will return a helpful error in case you omit this and attempt to upload a part with a partNumber of `0`.

In this example, for simplicity's sake, the data passed in to each call to `cos.doUploadPart` is the same empty 25 MB buffer.

After the array of promises is built, wait the resolution of every upload operation with `Promises.all`. As an `uploadPart` operation completes, it returns the calculated ETag value for the part that has been uploaded. These ETag values will be collected in the `parts` array prototype returned.

To complete the multipart upload operation, provide the `doCompleteMultiPartUpload` function with three parameters: The object key, the `UploadId`, and a list of parts and their ETags. The results of a `Promise.all` call are always arranged in the same order as the input array. You can use that to your advantage to iterate the returned `parts` array and add a `PartNumber` property to each entry.

When the `doCompleteMultiPartUpload` function completes successfully, it will return the following data:

- `Location`: The URI of the finalized object.
- `Bucket`: The bucket into which the object was uploaded.
- `Key`: The object's name on the Cloud Object Storage system.
- `ETag`: A hash useful for verifying object integrity (see below). This value will be suffixed with a dash (-), followed by a number. This number indicates the number of parts that make up the finalized object.

Run the code shown in Example 7-8.

Example 7-8 Creating multipart upload with key multipart.dat

```
~# node multipart_create_upload_and_complete.js

Creating multipart upload with key multipart.dat
{ Bucket: 'examplebucket',
  Key: 'multipart.dat',
  UploadId: '01000163-4661-1ff5-f666-0c27b4055ff9' }
Uploading part 1 for key multipart.dat with uploadId
01000163-4661-1ff5-f666-0c27b4055ff9
Uploading part 2 for key multipart.dat with uploadId
01000163-4661-1ff5-f666-0c27b4055ff9
Uploading part 3 for key multipart.dat with uploadId
01000163-4661-1ff5-f666-0c27b4055ff9
Uploading part 4 for key multipart.dat with uploadId
01000163-4661-1ff5-f666-0c27b4055ff9
Completing multipart upload for key multipart.dat with uploadId
01000163-4661-1ff5-f666-0c27b4055ff9
```

```
{ Location:
'https://examplebucket.s3.us-south.objectstorage.softlayer.net/multipart.dat',
  Bucket: 'examplebucket',
  Key: 'multipart.dat',
  ETag: '"99ec0e49c289164f03024888d45b78a8-4"' }
```

The multipart upload begins, immediately followed by all four part uploads. Note that these part uploads are taking place concurrently, which is the first benefit of multipart upload. After all four parts have uploaded, the script completes the upload, providing the Cloud Object Storage endpoint with a list of parts and their ETags.

Occasionally, you need to have direct control of each phase of the multipart upload operation. The functions and concepts discussed above form the building blocks of such a flow. However, more often you probably want to take advantage of the efficiencies provided by multipart upload without dealing with the additional complexity involved. The SDK provides just such a mechanism: The `ManagedUpload` class.

Multipart upload (with ManagedUpload)

`ManagedUpload` provides a simple API for uploading content to Cloud Object Storage that uses multipart uploads to achieve enhanced throughput, performance, and reliability.

When possible, `ManagedUpload` attempts to upload multiple parts of a single upload at once. When dealing with large file sizes and high bandwidth, this process can result in a significant increase in throughput. The `ManagedUpload` class defaults to a 5 MB part size and four concurrent uploads. Both of these defaults can be overridden as shown below.

`ManagedUpload` handles much of this complexity, including dividing objects into multiple parts, aborting failed uploads, returning progress events, adjusting the algorithm for maximum performance, and presenting a simple API for ease of use.

To see `ManagedUpload` at work, add a pair of functions to your Cloud Object Storage service module (`cos_advanced.js`). Open `cos_advanced.js` and update the “Public methods” and “Private methods” sections to include the code in Example 7-9.

Example 7-9 Updates to cos_advanced.js

```
// Public methods
module.exports.doBasicManagedUpload = _doBasicManagedUpload
module.exports.doManagedUpload = _doManagedUpload

// Private methods
// perform a basic ManagedUpload
function _doBasicManagedUpload(key, data) {
  console.log('Performing a managed upload of key', key);
  let params = {
    Bucket: _bucketName,
    Key: key,
    Body: data
  };
  return new Promise(function(resolve, reject) {
    cos.upload(params, function(err, data) {
      if (err) {
        reject(err)
      } else {
        resolve(data)
      }
    });
  });
}

// perform a basic ManagedUpload
function _doManagedUpload(key, data, partSize, concurrency) {
  console.log('Performing a managed upload of key', key);
  let params = {
    Bucket: _bucketName,
    Key: key,
    Body: data
  };
  let options = {
    partSize: partSize,
    queueSize: concurrency
  };
  return new Promise(function(resolve, reject) {
    cos.upload(params, options, function(err, data) {
      if (err) {
        reject(err)
      } else {
        resolve(data)
      }
    });
  });
}
```

You have added two functions to your growing Cloud Object Storage service module: `_doManagedUpload` and `_doBasicManagedUpload`. Both functions use the `ManagedUpload` class, with the "basic" function sticking to the 5MB/4 thread defaults mentioned previously. The `_doManagedUpload` function accepts additional parameters, allowing you to specify custom part size and concurrency numbers.

Next, create a file named `basic_managed_upload.js` and enter the code shown in Example 7-10.

Example 7-10 Code for basic_managed_upload.js

```
const cos = require('./cos_advanced');
let key = 'managedupload.dat'; // the object name
let buffer = new Buffer(1024 * 1024 * 25); // an empty 25MB buffer

cos.doBasicManagedUpload(key, buffer).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error)
});
```

Here, create an empty 25 MB buffer, and call the new `doBasicManagedUpload` function to upload the data to Cloud Object Storage.

Issue the command shown in Example 7-11.

Example 7-11 ~# node basic_managed_upload.js command

```
~# node basic_managed_upload.js
Performing a managed upload of key managedupload.dat
{ Location:
'https://examplebucket.s3.us-south.objectstorage.softlayer.net/managedupload.dat',
  Bucket: 'examplebucket',
  Key: 'managedupload.dat',
  ETag: '"01b86f5f0dc515188d548e2840c8bf40-5"' }
```

As seen in the output, the object was successfully uploaded to the Cloud Object Storage. By examining the returned ETag, specifically its “-5” suffix, you see that the object was, indeed, uploaded as five separate 5 MB parts.

Typically, when working with object sizes that can benefit from multipart upload, a part size in the 20 MB - 100 MB range is preferred. Create a file named `managed_upload.js` and enter the code shown in Example 7-12.

Example 7-12 Code for managed_upload.js

```
const cos = require('./cos_advanced');
let key = 'managedupload.dat'; // the object name
let buffer = new Buffer(1024 * 1024 * 120); // an empty 120MB buffer

cos.doManagedUpload(key, buffer, 1024 * 1024 * 20, 6).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error)
});
```

The call to `doManagedUpload` looks very similar to the `doBasicManagedUpload` you just performed. The important bits are found in the last two parameters passed to the function. You are now specifying the part size and concurrency (`queueSize`) for the multipart upload operation.

Issue the `managed_upload.js` command, as shown in Example 7-13.

Example 7-13 node managed_upload.js command

```
~# node managed_upload.js
Performing a managed upload of key managedupload.dat
{ Location:
'https://examplebucket.s3.us-south.objectstorage.softlayer.net/managedupload.dat',
  Bucket: 'examplebucket',
  Key: 'managedupload.dat',
  ETag: '"64f8e7993ee4d8c62b3d5cb2b26d40ab-6"' }
```

As shown in the output, the 120 MB object was uploaded as six separate 20 MB parts.

Note that the `ManagedUpload` class is not restricted to large objects that require multipart upload. Create a file named `managed_upload_non_multipart.js` and enter the code in Example 7-14.

Example 7-14 Code for managed_upload_non_multipart.js

```
const cos = require('./cos');
let key = 'managedupload.dat'; // the object name
let buffer = new Buffer(1024 * 1024 * 2); // an empty 2MB buffer

cos.doBasicManagedUpload(key, buffer).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error)
});
```

Here, you pass a 2 MB empty buffer to your `doBasicManagedUpload` function, which is well below the 5 MB threshold for multipart upload.

Issue the command shown in Example 7-15.

Example 7-15 managed_upload_non_multipart.js command

```
~# node managed_upload_non_multipart.js
Performing a managed upload of key managedupload.dat
{ ETag: '"b2d1236c286a3c0704224fe4105eca49"',
  Location:
'https://examplebucket.s3.us-south.objectstorage.softlayer.net/managedupload.dat',
  key: 'managedupload.dat',
  Key: 'managedupload.dat',
  Bucket: 'examplebucket' }
```

Examining the ETag from the result of the upload, you can see no sign of a `-#` suffix indicating multipart upload. The object was uploaded in its entirety in a single part.

This output shows the real power of the `ManagedUpload` class: With one function definition, you can pass in a string, buffer, or stream 10 bytes or 10 GB in size. You can rest assured that the data will be uploaded in an efficient manner. The class abstracts away the tedium of chunking incoming data, opening, maintaining, and cleaning up upload operations, but still affords the developer control over part size and concurrency.

When it comes time to choose an upload technique for your application, consider this general rule: Begin building with `ManagedUpload`. What begins as a basic use case that could be serviced just as well by `putObject` requests might grow to include larger, more complex data sets. Starting with the `MangedUpload` implementation allows your application to accommodate these new data sets effectively and efficiently with no change to your service module.

7.3.2 Downloading data from Cloud Object Storage

This section covers methods for getting data from Cloud Object Storage. Open `cos_advanced.js` and update the “Public methods” and “Private methods” sections to include the code in Example 7-16.

Example 7-16 Code for cos_advanced.js

```
// Public methods
module.exports.doReadObject = _doReadObject;
module.exports.doReadObjectRange = _doReadObjectRange;

// Private methods
// Read Examples
function _doReadObject(key) {
  console.log('Getting object with key', key);
  return new Promise(function(resolve, reject) {
    cos.getObject({
      Bucket: _bucketName,
      Key: key
    }, function(err, data) {
      if(err) reject(err);
      resolve(data);
    });
  });
}

function _doReadObjectRange(key, start_byte, end_byte) {
  console.log('Getting object with key', key);
  return new Promise(function(resolve, reject) {
    cos.getObject({
      Bucket: _bucketName,
      Key: key,
      Range: "bytes=" + start_byte + "-" + end_byte
    }, function(err, data) {
      if(err) reject(err);
      resolve(data);
    });
  });
}
```

You have added two functions to your service module: `_doReadObject` and `_doReadObjectRange`.

Standard download

The `_doReadObject` accepts a single argument, the key of the object to download from Cloud Object Storage. It then returns a promise that calls `getObject` on the given key and contains the data downloaded or an error if the operation fails.

This pattern (issuing a single GET on an object) is the most common technique for downloading data from Cloud Object Storage and is the method used throughout this tutorial series.

To see it in action, create a file named `basic_download.js` and enter the code shown in Example 7-17.

Example 7-17 Code for basic_download.js

```
const cos = require('./cos_advanced');
let key = 'managedupload.dat'; // the object name

// download the managedupload.dat object
cos.doReadObject(key).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error);
})
```

In this example, we will be downloading the 2 MB `managedupload.dat` object created at the end of the `ManagedUpload` section. If you have removed that object from your bucket, upload a new one or change the key to one that exists in your bucket.

Next, run the code shown in Example 7-18.

Example 7-18 The node basic_download.js command

```
~# node basic_download.js
Getting object with key managedupload.dat
{ AcceptRanges: 'bytes',
  LastModified: 'Fri, 11 May 2018 04:38:07 GMT',
  ContentLength: '2097152',
  ETag: '"b2d1236c286a3c0704224fe4105eca49"',
  ContentType: 'application/octet-stream',
  Metadata: {},
  Body: <Buffer 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00 ... > }
```

In the output, you can see that the download operation returns the object's metadata from the Cloud Object Storage system, along with the object's data in the Body field. You can take this returned Body buffer and perform whatever operation you would like on the data.

Range read

Occasionally, you do not need the whole object. In those cases, downloading its entirety from Cloud Object Storage would be a waste of bandwidth and, just as importantly, time.

If, for instance, you have a MapReduce job set up to spawn a number of workers to ingest a portion of an object and act on that data, sending the entirety of the object to each worker would be incredibly inefficient. Perhaps the object instead contains video data from a sporting event, but you are only interested in the highlights. Again, downloading the entire object would waste bandwidth and slow the user experience.

Luckily, the Cloud Object Storage SDK allows you to pass in an optional Range parameter to your call to `getObject` to only request the specific data that you are interested in.

Create a new file called range_download.js and enter the code shown in Example 7-19.

Example 7-19 Code for range_download.js

```
const cos = require('./cos_advanced');
let key = 'managedupload.dat'; // the object name
let begin_byte = 0; // the first byte to request
let end_byte = 9; // the last byte to request

// request the first 10 bytes from the managedupload.dat object
cos.doReadObjectRange(key, 0, 9).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error);
})
```

Here, you specify two new variables to specify the first and last bytes of the range you request: begin_byte and end_byte to specify the first and last bytes of the range you request. These variables will be passed to the doReadObjectRange function declared in your Cloud Object Storage service module where they will be used to construct a Range parameter for your getObject request. The Range parameter is a string, and, in the case of this example will be formatted in the request as follows: “bytes=0-9”.

Issue the command shown in Example 7-20.

Example 7-20 range_download.js command

```
~# node range_download.js
Getting object with key managedupload.dat
{ AcceptRanges: 'bytes',
  LastModified: 'Fri, 11 May 2018 04:38:07 GMT',
  ContentLength: '10',
  ETag: '"b2d1236c286a3c0704224fe4105eca49"',
  ContentRange: 'bytes 0-9/2097152',
  ContentType: 'application/octet-stream',
  Metadata: {},
  Body: <Buffer 00 00 00 00 00 00 00 00 00 00> }
```

The output shows that the ContentLength of the response was 10 (bytes). Compare this to the result of your previous basic_download.js run in which you downloaded the full object - ContentLength, which was 2097152. The range download only returned the 10 bytes you requested, saving you significant bandwidth and time.

Additionally, a new ContentRange field was returned with this request, reminding you of the byte range that you requested, along with the total size in bytes of the object.

7.3.3 Verifying data integrity

This module has covered several methods for storing data in and retrieving data from IBM Cloud Object Storage. However, how can you be confident that the data you are uploading and downloading has not been corrupted in transit?

An object's ETag metadata field, returned by many Cloud Object Storage SDK operations, allows you to verify the contents of an object by comparing it to a locally calculated value. If the two match, you can be confident that the data that you have on the local system is the same as the object on the Cloud Object Storage system.

Single part ETag calculation

At its simplest, a single-part object's ETag is the hexadecimal digest of the data's md5 checksum. You can verify this easily by creating a file named `basic_etag.js` and enter the code shown in Example 7-21.

Example 7-21 The basic_etag.js command

```
const cos = require('./cos');
let key = 'managedupload.dat'
let buffer = new Buffer(1024 * 1024 * 2);
let hashBuffer = crypto.createHash('md5').update(buffer).digest('hex');

cos.doBasicManagedUpload(key, buffer).then(function(res) {
  console.log(res);
  console.log('md5sum of data:', hashBuffer.toString('utf8'));
}).catch(function(error) {
  console.error(error)
});
```

This code uses your basic `ManagedUpload` code to upload an empty 2 MB buffer to the object storage system. Here, after creating the buffer, calculate its `md5sum` to later compare against the ETag that will be returned after the upload completes.

Issue the command shown in Example 7-22.

Example 7-22 The node basic_etag.js command

```
~# node basic_etag.js
Performing a managed upload of key managedupload.dat
{ ETag: '"b2d1236c286a3c0704224fe4105eca49"',
  Location:
'https://examplebucket.s3.us-south.objectstorage.softlayer.net/managedupload.dat',
  key: 'managedupload.dat',
  Key: 'managedupload.dat',
  Bucket: 'examplebucket' }
md5sum of data: b2d1236c286a3c0704224fe4105eca49
```

As the output shows, the `md5sum` of your local buffer is the same as the ETag returned by the Cloud Object Storage upload operation.

Multipart ETag Calculation

You can similarly calculate the ETag of an object that was uploaded with multipart upload. This process involves a few extra steps because the checksum calculation is not performed against the full, finalized object. Rather, it is performed on each of its constituent parts and then again on the concatenated binary form of all those hashes.

To verify the ETag of an object uploaded with multipart upload, complete these steps:

1. Calculate the md5 hash for each uploaded part of the object.
2. Concatenate those hashes into a single binary.
3. Calculate the md5 hash of the resulting binary.
4. Get the hex digest of the hash from step 3.

To see this in action, create a 100 MB file to work with by issuing this command:

```
~# dd if=/dev/zero of=100mb.dat bs=1m count=100
```

Next, check the md5 sum of the new file:

```
~# md5 100mb.dat
     MD5 (100mb.dat) = 2f282b84e7e608d5852449ed940bfc51
```

Create a file named `upload_file.js` and enter the code shown in Example 7-23.

Example 7-23 Code for upload_file.js

```
const cos = require('./cos_advanced');
const fs = require('fs');
let buffer = fs.readFileSync('100mb.dat'); // read the 100mb.dat file into a 
buffer
let key = '100mb.dat'; // the object name

// upload the buffer (the 100mb.dat file) with a part size of 10MB, and 
concurrency of 5 threads
cos.doManagedUpload(key, buffer, 1024 * 1024 * 10, 5).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error)
});
```

Use the `ManagedUpload` class discussed previously to upload your 100 MB file to Cloud Object Storage in 10 MB parts with five concurrent uploads.

Issue the `upload_file.js` command, as shown in Example 7-24.

Example 7-24 upload_file.js command

```
~# node upload_file.js
Performing a managed upload of key 100mb.dat
{ Location: 
'https://examplebucket.s3.us-south.objectstorage.softlayer.net/100mb.dat',
  Bucket: 'examplebucket',
  Key: '100mb.dat',
  ETag: '"1761756dcf22c0947a9b6e626b409659-10"' }
```

The returned result shows that the ETag does not match the simple md5 sum you calculated before the upload. Because you know the part count and part size settings used to upload the object, you can recreate the multipart ETag. Keep the `100mb.dat` file around because you will need it in a moment.

Open `cos_advanced.js` and update the “Public methods” and “Private methods” sections to include the code shown in Example 7-25.

Example 7-25 Code for cos_advanced.js

```
// Public methods
module.exports.doHashMultipartFile = _doHashMultipartFile;
module.exports.checkEtagForMultipart = _checkEtagForMultipart;
module.exports.calculatePartSize = _calculatePartSize;

// Private methods
// ETag Examples
// read a file and return a promise to calculate the ETag, given a number of parts 
and part size
function _doHashMultipartFile(file, parts, partSize) {
  console.log('Calculating Multipart ETag');
  return new Promise(function (resolve, reject) {
    var hashBuffer = new Buffer(0); // create an empty buffer to hold calculated
hashes
    var readStream = fs.createReadStream(file, { highWaterMark: partSize });
    var hash = ''

    readStream.on('data', function(chunk) {
      // concatenate the md5sum of this chunk to the hashBuffer
      hashBuffer = Buffer.concat([hashBuffer,
crypto.createHash('md5').update(chunk).digest()]);
    }).on('end', function() {
      // calculate the md5 hash of the full hashBuffer, then get the hex digest
      hash = crypto.createHash('md5').update(hashBuffer).digest('hex');
      resolve(hash + "-" + parts.toString())
    });
  })
}

// check an ETag for multipart ('-###') suffix and return a promise
function _checkEtagForMultipart(etag) {
  console.log('Checking ETag for multipart upload')
  return new Promise(function (resolve, reject) {
    var res = etag.match(/-\d+$/); // regex match the "-###" suffix from a
multipart upload etag
    if (res != null) {
      resolve(parseInt(res[0].substr(1))) // return the number of parts if found
    } else {
      resolve(null) // return null if no multipart suffix is found
    }
  })
}

// calculate the part size of a multipart file in bytes
function _calculatePartSize(file, parts) {
  console.log('Calculating multipart size for ' + file);
  var file_size = fs.statSync(file).size; // get the size of a file on disk
  var split = Math.floor(file_size / parts); // divide the file_size by the number
of parts, rounding the result down
  var megabytePartSize = Math.ceil(split / (1024 * 1024)); // divide the 'split'
size by 1MB, rounding the result up
  return Math.floor(megabytePartSize * (1024 * 1024)); // return the part size (in
bytes), rounding down
}
```

Several functions have been added. Start by looking at `_doHashMultipartFile`, which takes the parameters shown in Table 7-4.

Table 7-4 Parameters of _doHashMultipartFile

Parameter	Type	Description
file	String	The name of a file on disk
parts	Integer	The number of parts used in a multipart upload
partSize	Integer	The size of parts used in a multipart upload in bytes

The new function opens the given file, reads it in `partSize` chunks, hashes those chunks, and appends the result to a buffer (`hashBuffer`). After the file is completely read, the function calculates the hex digest of the full `hashBuffer`, then appends the parts suffix (`-#`) to the result and returns it.

Create a file named `hash_multipart_file.js` and enter the code in Example 7-26.

Example 7-26 Code for hash_multipart_file.js

```
const cos = require('./upload_working');
const fs = require('fs');

cos.doHashMultipartFile('100mb.dat', 10, 10 * 1024 * 1024).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error);
})
```

Call your new `doHashMultipartFile` function, passing in the `100mb.dat` file created above, a part count (`parts`) of `10`, and a `partSize` of `10 MB` (10 * 1024 * 1024).

Issue the **`hash_multipart_file.js`** command, as shown in Example 7-27.

Example 7-27 ash_multipart_file.js command

```
~# node hash_multipart_file.js
Calculating Multipart ETag
1761756dcf22c0947a9b6e626b409659-10
```

As shown in the output, the ETag calculation now matches what was returned earlier when you uploaded the object. With this in hand, verify that the object uploaded to Cloud Object Storage matches your local copy so you can be confident in the integrity of your uploaded data.

Similarly, you can verify the integrity of data downloaded from the Cloud Object Storage system. Because you will not always know the part size used to create the original upload, you can take the ETag returned from the Cloud Object Storage system and use math to recreate the ETag as calculated on the object storage system.

To do so, you need the other two functions you have added to `cos_advanced.js` in this section. First, examine `_checkEtagForMultipart`. This function accepts a single parameter (an ETag) and uses a regular expression match to determine whether it contains a multipart suffix (a dash, followed by a number of digits, such as `-123`). It returns the part count as an integer if it finds one. Otherwise, it returns `null`.

Now that you can reliably pull the part count from any ETag, use that information to calculate part size using `_calculatePartSize`. This function accepts a file name as a string and an integer part count. The function then gets the file size from the local file system, and divides that byte number by the part count, rounding down. It then converts the result (in bytes) to megabytes, rounding up, before re-converting to bytes, rounded down, and returning the result. Because the last part in a multipart upload is rarely the same size as the rest of the parts of the upload, this seemingly convoluted math ensures that you return the part size used for all parts except for the final one.

You now have all that you need to calculate the ETag: A file, its part size, and part count. Pass each into the `_doHashMultipartFile` function to put your knowledge to the test.

First, create a file:

```
dd if=/dev/random of=20mb.dat bs=1m count=20
```

Then, create a file named `20mb_upload.js` and enter the code in Example 7-28.

Example 7-28 Code for 20mb_upload.js

```
const cos = require('./cos_advanced');
const fs = require('fs');
let key = '20mb.dat'; // the object name
let parts = 0;
let partSize = 0;
let buffer = fs.readFileSync(key);

cos.doBasicManagedUpload(key, buffer).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error)
});
```

This should look familiar. You are uploading the object using the `ManagedUpload` class and default parameters (5 MB part size).

Create a file named `download_and_calculate_etag.js` and enter the code shown in Example 7-29.

Example 7-29 Code for download_and_calculate_etag.js

```
const cos = require('./cos_advanced');
const fs = require('fs');
let key = '100mb.dat'; // the object name
let parts = 0;
let partSize = 0;

// download the 100mb.dat object
cos.doReadObject(key).then(function(res) {
  console.log("ETag from COS",res.ETag);
  fs.writeFile(key, res.Body, function(err) {
    if (err) {
      console.error(err);
    }
  });
  return res;
}).then(function(response) {
  parts = cos.checkEtagForMultipart(response.ETag); // get the part count from the
ETag
  return parts;
}).then(function(partCount) {
  if (partCount) {
    partSize = cos.calculatePartSize(key, partCount);
  }
}).catch(function(error) {
  console.error(error);
});
```

You are using a number of functions here. Because each function returns a promise, chain them together with `.then` clauses. The code first reads back the object from Cloud Object Storage. When this step resolves, the code returns the response and passes it to the function that checks the returned ETag for a multipart upload suffix. If that suffix is found, the script returns the part count and pass it to your function to calculate part size.

Issue the command shown in Example 7-30.

Example 7-30 The download_and_calculate_parts.js command

```
~# node download_and_calculate_parts.js
Getting object with key 100mb.dat
ETag from COS "1761756dcf22c0947a9b6e626b409659-10"
Checking ETag for multipart upload
Found 10 parts
Calculating multipart size for 100mb.dat
Calculated part size as 10485760
```

You can take what you learned about the object and use the `doHashMultipartFile` function to verify the data you have downloaded. Create a file named `doublecheck_etag.js` and enter the code shown in Example 7-31.

Example 7-31 Code for doublecheck_etag.js

```
const cos = require('./upload_working');
const fs = require('fs');

cos.doHashMultipartFile('100mb.dat', 10, 10485760).then(function(res) {
  console.log(res);
}).catch(function(error) {
  console.error(error);
})
```

Issue the command shown in Example 7-32.

Example 7-32 The doublecheck_etag.js command

```
~# node doublecheck_etag.js
Calculating Multipart ETag of 100mb.dat
Part Size: 10485760, Part Count: 10
1761756dcf22c0947a9b6e626b409659-10
```

The ETag that is calculated matches the value stored in Cloud Object Storage, so you and can be confident that the data you received is intact.

7.3.4 Presigned URLs

IBM Cloud Object Storage provides a full complement of security and access management mechanisms to ensure that users and roles associated with an IBM Cloud account have appropriate levels of access to objects stored therein.

Developers commonly need to make IBM Cloud Object Storage resources (buckets or objects) available outside of the account, either to everyone or just to authenticated/trusted users.

If you need to make an object truly public (available for anyone to issue a `HEAD` or `GET` request against it), setting the ACL on the object is the simplest solution. This setting can be done either when creating an object or after the object has been uploaded, and can also be reverted at any time. See the Allowing public access section for details.

Often, making an object completely public is not what you want to do. For example, you might want an object to be available to download only after a user has successfully logged into your application. You might also need to provide your users a location to which they can upload data without giving the entire world public write access to your Cloud Object Storage storage.

In both cases, you need a mechanism for keeping your Cloud Object Storage resources private, but providing access to others on occasion. Presigned URLs grant you this flexibility. They provide developers with the ability to create and share a unique URL with access to a single resource, and to ensure that the URL has a finite lifetime.

Open `cos_advanced.js` and update the “Public methods” and “Private methods” sections to include the code in Example 7-33.

Example 7-33 Code to add to cos_advanced.js

```
// Public methods
module.exports.generatePresignedGet = _generatePresignedGet;
module.exports.generatePresignedGetWithExpiration =
_generatePresignedGetWithExpiration;
module.exports.generatePresignedPut = _generatePresignedPut;
module.exports.generatePresignedPutWithExpiration =
_generatePresignedPutWithExpiration;

// Private methods
// create a presigned GET URL with default 900 second expiry
function _generatePresignedGet(key) {
  console.log('Generating presigned GET URL for key', key);
    var url = cos.getSignedUrl('getObject', {
      Bucket: _bucketName,
      Key: key
    });
    console.log(url);
}

// create a presigned GET URL with an expiration in seconds
function _generatePresignedGetWithExpiration(key, expiration) {
  console.log('Generating presigned GET URL for key ' + key + ' with expiration '
+ expiration);
    var url = cos.getSignedUrl('getObject', {
      Bucket: _bucketName,
      Key: key,
      Expires: expiration
    });
    console.log(url);
}

// create a presigned PUT URL with default 900 second expiry
function _generatePresignedPut(key) {
  console.log('Generating presigned PUT URL for key', key);
  var params = {
    Bucket: _bucketName,
    Key: key
  };
  cos.getSignedUrl('putObject', params, function (err, url) {
    console.log(url);
  });
}

// create a presigned PUT URL with an expiration in seconds
function _generatePresignedPutWithExpiration(key, expiration) {
  console.log('Generating presigned PUT URL for key ' + key + ' with expiration '
+ expiration);
  var params = {
    Bucket: _bucketName,
    Key: key,
    Expires: expiration
  };
  cos.getSignedUrl('putObject', params, function (err, url) {
    console.log(url);
  });
}
```

This code creates four new functions at the end of the file: `_generatePresignedGet`, `_generatePresignedGetWithExpiration`, `_generatePresignedPut`, and `_generatePresignedPutWithExpiration`. In production code, these could be consolidated down to a single function with additional parameters, but are separated here for clarity.

The two “basic” functions (`_generatePresignedGet` and `_generatePresignedPut`) both accept a key parameter (the object name to be read or uploaded) and use the default `expiry` (900 seconds / 15 minutes) value. The “WithExpiration” functions both accept a `key` parameter and an additional integer expiration that will set the `expiry` in seconds.

Create a file named `presigned_urls.js` and complete it to match the code shown in Example 7-34.

Example 7-34 Code for presigned_urls.js

```
const cos = require('./cos_advanced');
const fs = require('fs');
let getKey = '100mb.dat'; // the object name
let putKey = 'helloIBM';

cos.generatePresignedGet(getKey);
cos.generatePresignedGetWithExpiration(getkey, 3000);
cos.generatePresignedPut(putKey);
cos.generatePresignedPutWithExpiration(putKey, 27000);
```

Issue the commands shown in Example 7-35.

Example 7-35 The presigned_urls.js command

```
~# node presigned_urls.js
Generating presigned GET URL for key 100mb.dat
https://examplebucket.s3.us-south.objectstorage.softlayer.net/100mb.dat?AWSAccessK
eyId=................................&Expires=1526619000&Signature=hpwNvlhZRxhIKih
vFet%2Fu9gDH8O%3D
```

```
Generating presigned GET URL for key 100mb.dat with expiration 3000
https://examplebucket.s3.us-south.objectstorage.softlayer.net/100mb.dat?AWSAccessK
eyId=..............................&Expires=1526618101&Signature=501TzZmJ34ULYCF
Kh29eLEvdV3I%3D

Generating presigned PUT URL for key helloIBM
https://examplebucket.s3.us-south.objectstorage.softlayer.net/helloIBM?AWSAccessKe
yId=..............................&Expires=1526619000&Signature=FN3VGGKnLe8HePdZ
1dEnZ0d1spM%3D

Generating presigned PUT URL for key helloIBM with expiration 27000
https://examplebucket.s3.us-south.objectstorage.softlayer.net/helloIBM?AWSAccessKe
yId=..............................&Expires=1526645100&Signature=YEGMA07ArObkbKhZ
p8GycXImRE8%3D
```

You will see different values for the `AWSAccessKeyID`, `Expires`, and `Signature` tokens, but if the code returns four URLs, you are successful. The first two URLS provide time-bound access to the `100mb.dat` object from the previous sections. Paste it into your browser or download it with `cURL/wget`. Notice that you are not prompted for any credentials because you have signed the URLs with Cloud Object Storage credentials ahead of time and can now share this URL however you need to.

Similarly, you can issue a PUT at either of the two final URLs returned by the code and write to the `helloIBM` object in your Cloud Object Storage account.

Using `curl` with a local file named `file.txt`, you can issue this command (replacing `$putURL` with the URL returned above):

```
curl -v -T file.txt $putURL
```

Again, the system does not prompt for credentials during the PUT operation because you have already signed the URL from your Cloud Object Storage account. As you develop your applications, use this mechanism to allow users to upload data directly to Cloud Object Storage without having to stage it on an intermediate server.

Examine what happens when your presigned URL expires by reopening `presigned_urls.js` and editing it to match the code shown in Example 7-36.

Example 7-36 Code to add to presigned_url.js

```
const cos = require('./cos_advanced');
const fs = require('fs');
let getKey = '100mb.dat'; // the object name

cos.generatePresignedGetWithExpiration(getkey, 1);
```

Run the code, wait 10 seconds, and then try to GET the URL that returns, as shown in Example 7-37.

Example 7-37 presigned_urls.js command

```
~# node presigned_urls.js
Generating presigned GET URL for key 100mb.dat with expiration 1
https://examplebucket.s3.us-south.objectstorage.softlayer.net/100mb.dat?AWSAccessK
eyId=..............................&Expires=1526619316&Signature=501TzZmJ34ULYCF
Kh29eLEvdV3I%3D
```

```
~# curl
https://examplebucket.s3.us-south.objectstorage.softlayer.net/100mb.dat?AWSAccessK
eyId=..............................&Expires=1526619316&Signature=501TzZmJ34ULYCF
Kh29eLEvdV3I%3D

<?xml version="1.0" encoding="UTF-8"
standalone="yes"?><Error><Code>ExpiredToken</Code><Message>The provided token has
expired.</Message><Resource>/100mb.dat</Resource><RequestId>67cf6b72-770a-4c93-ab2
a-5fe0e5317379</RequestId><httpStatusCode>403</httpStatusCode></Error>%
```

You see a `403 ExpiredToken` response from Cloud Object Storage. The presigned URL was valid for one second, then expired, effectively deactivating the URL. A one-second expiration clearly has limited real-world value, but this example shows a real benefit of using presigned URLs: After they reach the end of their lifetimes, they automatically close off access to the resource to which they were pointed.

Presigned URLs provide developers with a way to safely allow users to interact with Cloud Object Storage resources, providing time-bound access to specific operations and to manage the lifecycle of that access.

7.3.5 Tiers in Cloud Object Storage

IBM Cloud Object Storage offers various pricing models for storage of data with no performance penalties. IBM Cloud Object Storage offers the following tiers:

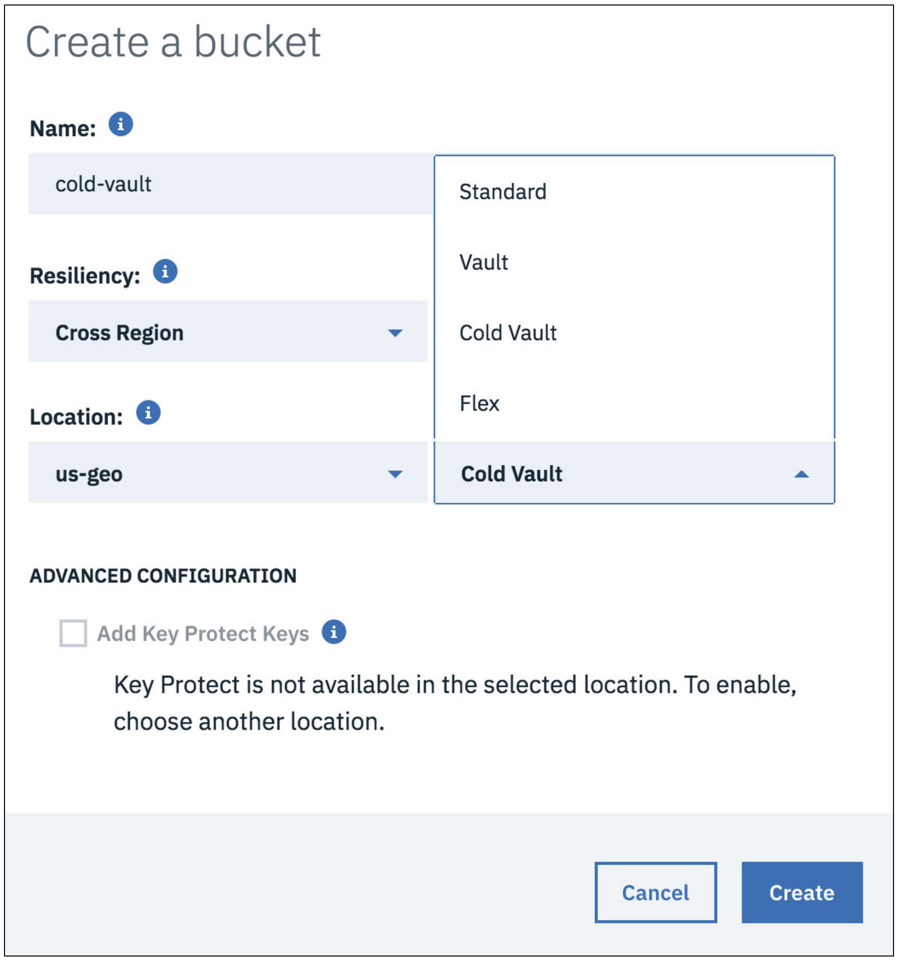
- Standard: Used for accessed multiple times per month.
- Vault: Used for less active data accessed once a month or less.
- Cold Vault: Used for rarely accessed data and long-term data retention.
- Flex: For dynamic data with varying month-to-month access needs, ideal for a mix of hot and cold workloads.

In the Insurance Company's X case, we chose Standard buckets for storing all the images. However, a better solution could be storing the original images uploaded by the user to Cold Vaults because they are rarely accessed. The processed images could be placed in Standard Vault because they might be frequently accessed.

The following are other cases where cold storage tiers could be useful:

- When claims are archived, it can go into the cold vault.
- Cloudant backups can be written to cold vault.

Using Cold Vault reduces the overall storage costs. Creating a Cold Vault is shown in Figure 7-1.


Create a bucket

Name: cold-vault

Resiliency: Cross Region

Location: us-geo

Cold Vault

ADVANCED CONFIGURATION

Add Key Protect Keys

Key Protect is not available in the selected location. To enable, choose another location.

Figure 7-1 Creating a Cold Vault

Complete the same procedure to create a Standard Vault.

Note the following about the vaults:

- You cannot move objects between tiers.
- You use the same access APIs to create, delete, and access objects.
- You will not notice any performance impacts when using any of the tiers

7.4 Next steps

After you complete this module, continue with the same code in Chapter 8, “Discover insights using Watson Services” on page 113.

7.5 Other references

See these websites for more information about Multipart Upload functions:

- The createMultipartUpload operation:

  https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#createMultipartUpload-property

- The uploadPart operation

  https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#uploadPart-property

- The abortMultipartUpload operation

  https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#abortMultipartUpload-property

- The completeMultipartUpload operation

  https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#completeMultipartUpload-property

8

Discover insights using Watson Services

After adding images to the workflow, a new world of capabilities using the current technologies are available. While traditional applications require input for all the registered information, the new era allows you to create much more data out of the existing data.

By introducing AI and elaborating on it in the next module, you can reduce the input from the user while also enhancing the application and making it more dynamic. This module demonstrates the abilities in the Watson Visual Recognition API that allow you to perform content validation and enhancement using the available data sources.

This chapter includes the following sections:

8.1 Current architecture

The current architecture already collects information from your application user and stores the information in your Cloudant database.

The following are steps in the validation of the object using the Watson Visual Recognition service:

1. Send the images to VR default classifiers to validate that the object is a car.
2. Send the images to VR default classifiers to validate that the car color matches the color in the insurance database.

The following are steps in the enhancement of the object using the Watson Visual Recognition service:

1. Instead of asking the client to drag the picture to the correct box, the VR service with a self-trained classifier will enhance/match the picture with the correct angle.
2. Review each image for damage and classify the damage using a self-trained classifier.

8.2 Learning objectives

In this module, you will:

- Understand the capabilities of the Watson Visual Recognition service:
  - Review the API explorer and use it without integration
  - Use the default classifier on a random picture
- Implement API within your node.js application:
  - Use the default classifier for content validation
  - Create your own classifier and call it from within the application
  - Enhance/retrain your own classifier and validate the results

8.3 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 7 to get the finished code from the previous module that you will build on:

```
$ cd module07
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module08
```

The IBM Watson Visual Recognition service uses deep learning algorithms to analyze images for scenes, objects, faces, and other content. The response includes keywords that provide information about the content. Researching how a deep learning algorithm understands your picture helps you when choosing images to train your algorithm.

It is important to have sufficient variation in your images. For example, when the army trained an algorithm to recognize tanks, they used hundreds of pictures to teach the algorithm. However, they then tried to demonstrate the algorithm during the winter and it did not work. This failure occurred because all of the training images involved different tanks, but they were all taken with a bright blue sky in the summer. The algorithm was unable to identify those same tanks in the winter.

8.3.1 Available models

A set of built-in models provides highly accurate results without training:

- General model: Default classification from thousands of classes.
- Face model: Facial analysis with age and gender.
- Explicit model (Beta): Whether an image is inappropriate for general use.
- Food model (Beta): Classification of images of food items.
- Text model (Private beta): Text extraction from natural scene images.

You can also train custom models to create specialized classes.

8.3.2 How to use the service

Figure 8-1 shows the process of creating and using Visual Recognition.

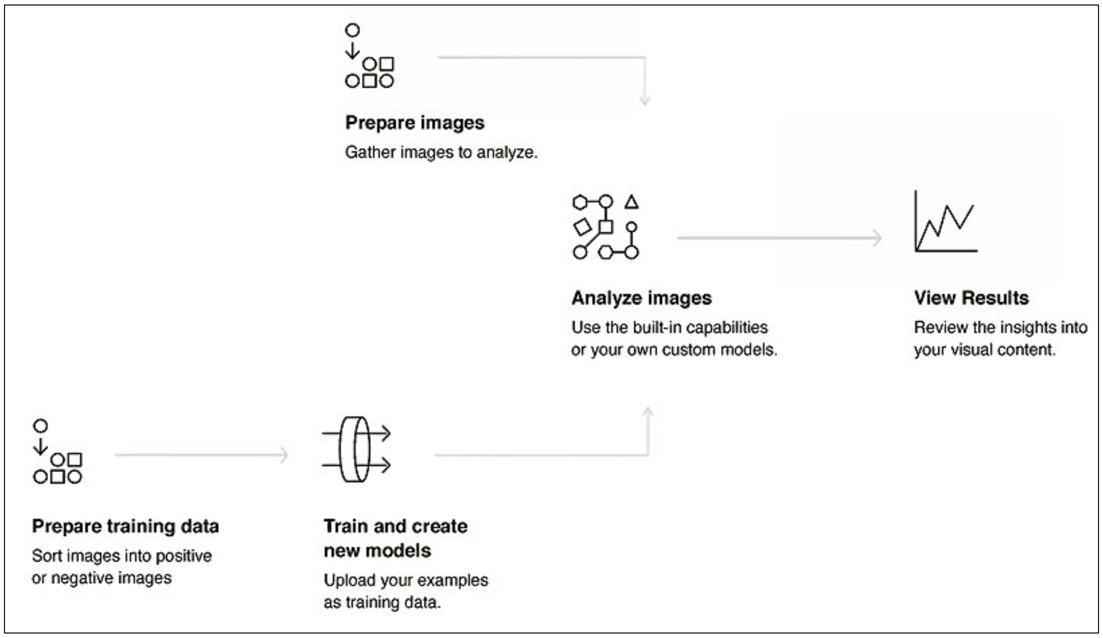

Prepare images
Gather images to analyze.
Analyze images
Use the built-in capabilities or your own custom models.
View Results
Review the insights into your visual content.
Prepare training data
Sort images into positive or negative images
Train and create new models
Upload your examples as training data.

Figure 8-1 Creating and using Visual Recognition

8.3.3 Use cases

The Visual Recognition service can be used for diverse applications and industries:

- Manufacturing: Use images from a manufacturing setting to make sure that products are being positioned correctly on an assembly line.
- Visual auditing: Look for visual compliance or deterioration in a fleet of trucks, planes, or windmills out in the field, and train custom models to understand what defects look like.

- Insurance: Rapidly process claims by using images to classify claims into different categories.
- Social listening: Use images from your product line or your logo to track buzz about your company on social media.
- Social commerce: Use an image of a plated dish to find out which restaurant serves it and find reviews, or use a travel photo to find vacation suggestions based on similar experiences.
- Retail: Take a photo of a favorite outfit to find stores with those clothes in stock or on sale, or use a travel image to find retail suggestions in that area.
- Education: Create image-based applications to educate about taxonomies.

8.4 Implementation

Perform the following steps to implement this solution.

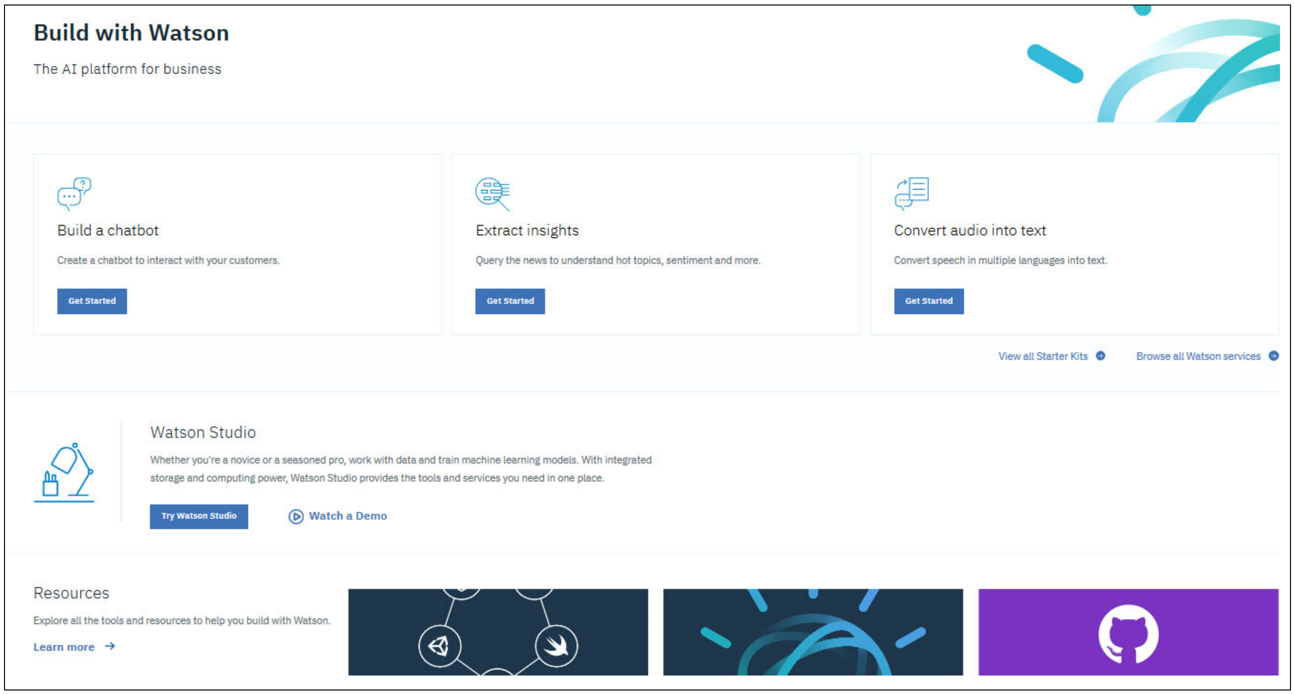
8.4.1 Local configuration, tool chains

This module uses the existing Watson Visual Recognition Service, so you do not need to spend time developing your own notebooks, but instead focus on the content. Everything is called from the API, which can be used in many different forms.

8.4.2 Provisioning steps for IBM Cloud services

With a free account, you are (at the time of the writing) allowed to have 250 API calls per day. This amount should be sufficient for the initial tests. However, it will only allow a single custom classifier, which creates some restrictions on the complete application. Therefore, you might want to create a paid account that will allow you to perform all the following functions.

Figure 8-2 shows the welcome page.



Build with Watson

The AI platform for business

Build a chatbot

Create a chatbot to interact with your customers.

Get Started

Extract insights

Query the news to understand hot topics, sentiment and more.

Get Started

Convert audio into text

Convert speech in multiple languages into text.

Get Started

View all Starter Kits

Browse all Watson services

Watson Studio

Whether you're a novice or a seasoned pro, work with data and train machine learning models. With integrated storage and computing power, Watson Studio provides the tools and services you need in one place.

Try Watson Studio

Watch a Demo

Resources

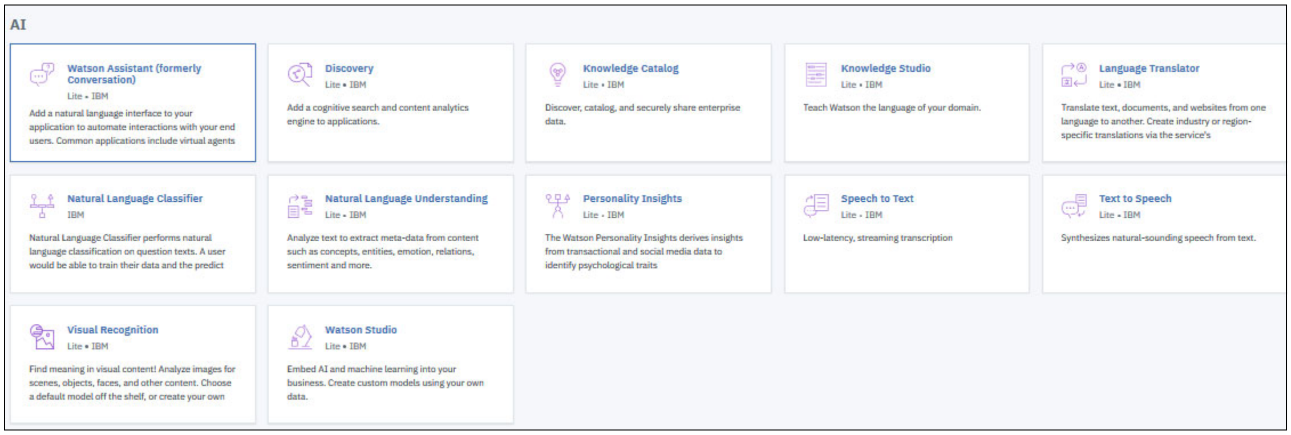
Explore all the tools and resources to help you build with Watson.

Learn more →

Figure 8-2   Build with Watson window

To create the Watson service API key, complete these steps:

1. Log in to your IBM Cloud account and select the correct space.
2. From the services, select **AI**, which will list all the available Watson services, as shown in Figure 8-3.



AI

Watson Assistant (formerly Conversation)
Lite • IBM
Add a natural language interface to your application to automate interactions with your end users. Common applications include virtual agents

Discovery
Lite • IBM
Add a cognitive search and content analytics engine to applications.

Knowledge Catalog
Lite • IBM
Discover, catalog, and securely share enterprise data.

Knowledge Studio
Lite • IBM
Teach Watson the language of your domain.

Language Translator
Lite • IBM
Translate text, documents, and websites from one language to another. Create industry or region-specific translations via the service's

Natural Language Classifier
IBM
Natural Language Classifier performs natural language classification on question texts. A user would be able to train their data and the predict

Natural Language Understanding
Lite • IBM
Analyze text to extract meta-data from content such as concepts, entities, emotion, relations, sentiment and more.

Personality Insights
Lite • IBM
The Watson Personality Insights derives insights from transactional and social media data to identify psychological traits

Speech to Text
Lite • IBM
Low-latency, streaming transcription

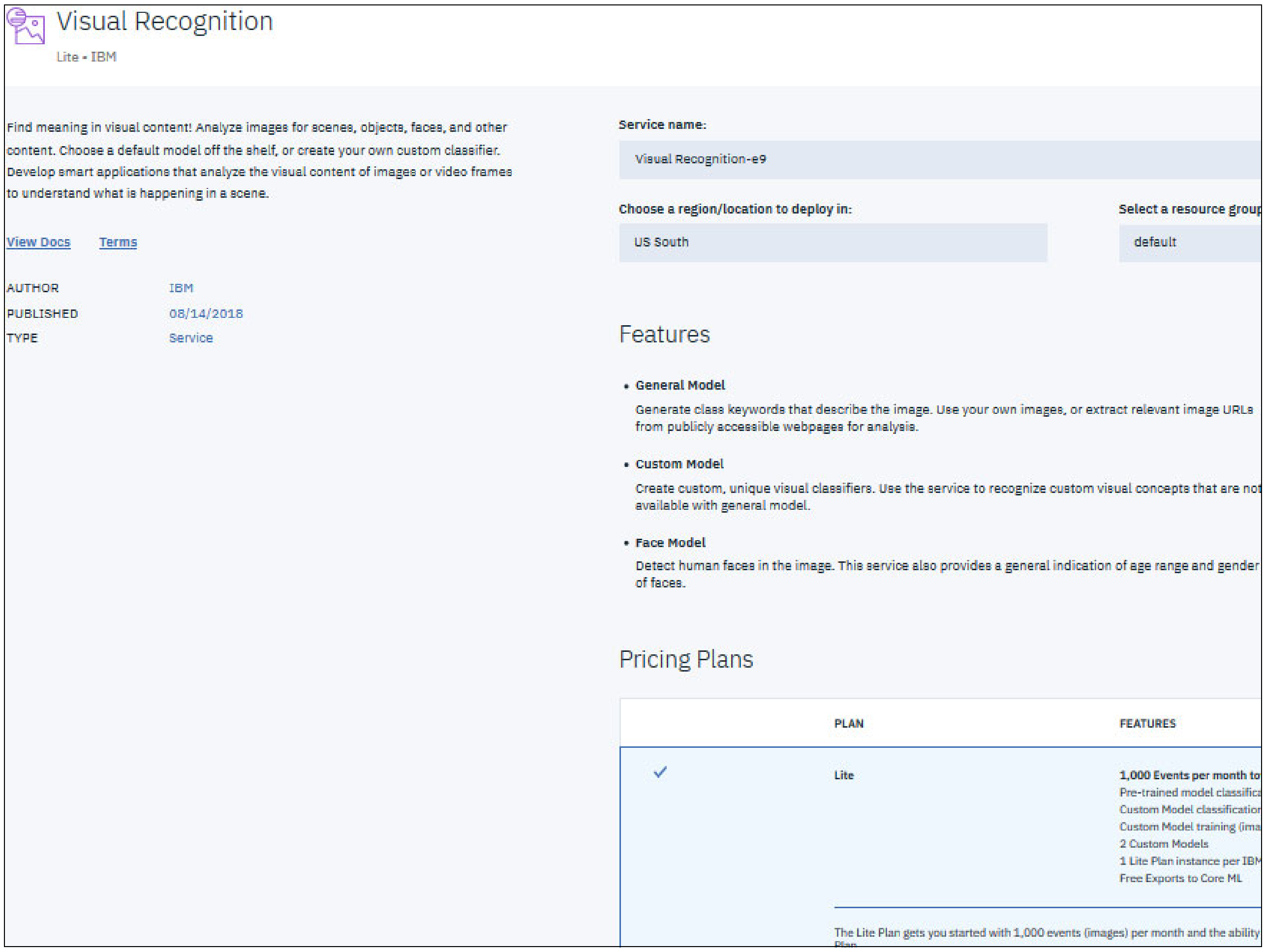
Text to Speech
Lite • IBM
Synthesizes natural-sounding speech from text.

Visual Recognition
Lite • IBM
Find meaning in visual content! Analyze images for scenes, objects, faces, and other content. Choose a default model off the shelf, or create your own

Watson Studio
Lite • IBM
Embed AI and machine learning into your business. Create custom models using your own data.

Figure 8-3   Available Watson services

3. This module focuses on the IBM Watson Visual Recognition service. Click the service to open the service details, as shown in Figure 8-4.



Visual Recognition

Lite • IBM

Find meaning in visual content! Analyze images for scenes, objects, faces, and other content. Choose a default model off the shelf, or create your own custom classifier. Develop smart applications that analyze the visual content of images or video frames to understand what is happening in a scene.

View Docs Terms

AUTHOR	IBM
PUBLISHED	08/14/2018
TYPE	Service

Service name: Visual Recognition-e9

Choose a region/location to deploy in: US South

Select a resource group: default

Features

- **General Model**
  Generate class keywords that describe the image. Use your own images, or extract relevant image URLs from publicly accessible webpages for analysis.
- **Custom Model**
  Create custom, unique visual classifiers. Use the service to recognize custom visual concepts that are not available with general model.
- **Face Model**
  Detect human faces in the image. This service also provides a general indication of age range and gender of faces.

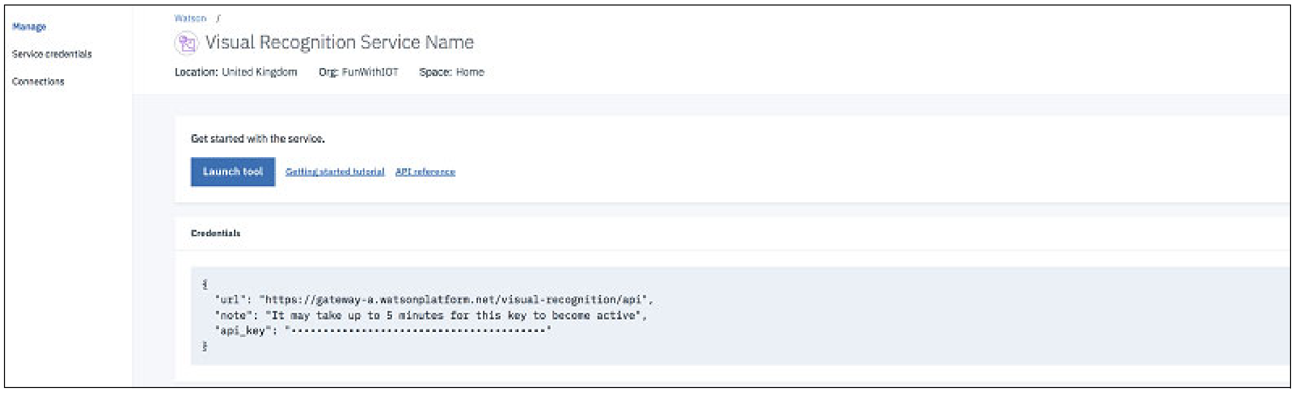
Pricing Plans

	PLAN	FEATURES
✓	Lite	**1,000 Events per month to** Pre-trained model classifica Custom Model classificatior Custom Model training (ima 2 Custom Models 1 Lite Plan instance per IBM Free Exports to Core ML

The Lite Plan gets you started with 1,000 events (images) per month and the ability

Figure 8-4 Service details of Watson Visual Recognition

4. As explained, it is possible to start with the Lite plan. However, you can select the standard plan at low cost to allow you to use more complex classifiers, as shown in Figure 8-5.



Manage
Service credentials
Connections

Watson /

Visual Recognition Service Name

Location: United Kingdom Org: FunWithIOT Space: Home

Get started with the service.

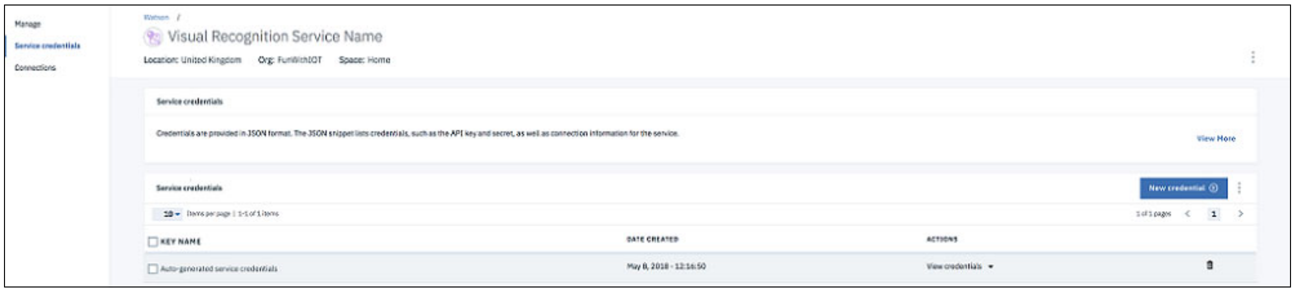
Launch tool Getting started tutorial API reference

Credentials

```
{
  "url": "https://gateway-a.watsonplatform.net/visual-recognition/api",
  "note": "It may take up to 5 minutes for this key to become active",
  "api_key": "......................................."
}
```

Figure 8-5 Selecting the plan

5. When you create a service, the first API key is automatically generated. This key must be used within your application to call the Watson Visual Recognition Service. All your monthly spend and details can be found in this overview, as shown in Figure 8-6.



Visual Recognition Service Name

Location: United Kingdom Org: FunWithIOT Space: Home

Service credentials

Credentials are provided in JSON format. The JSON snippet lists credentials, such as the API key and secret, as well as connection information for the service.

Service credentials

10 Items per page | 1-1 of 1 items   1 of 1 pages 1

KEY NAME	DATE CREATED	ACTIONS
Auto-generated service credentials	May 8, 2018 - 12:16:50	View credentials

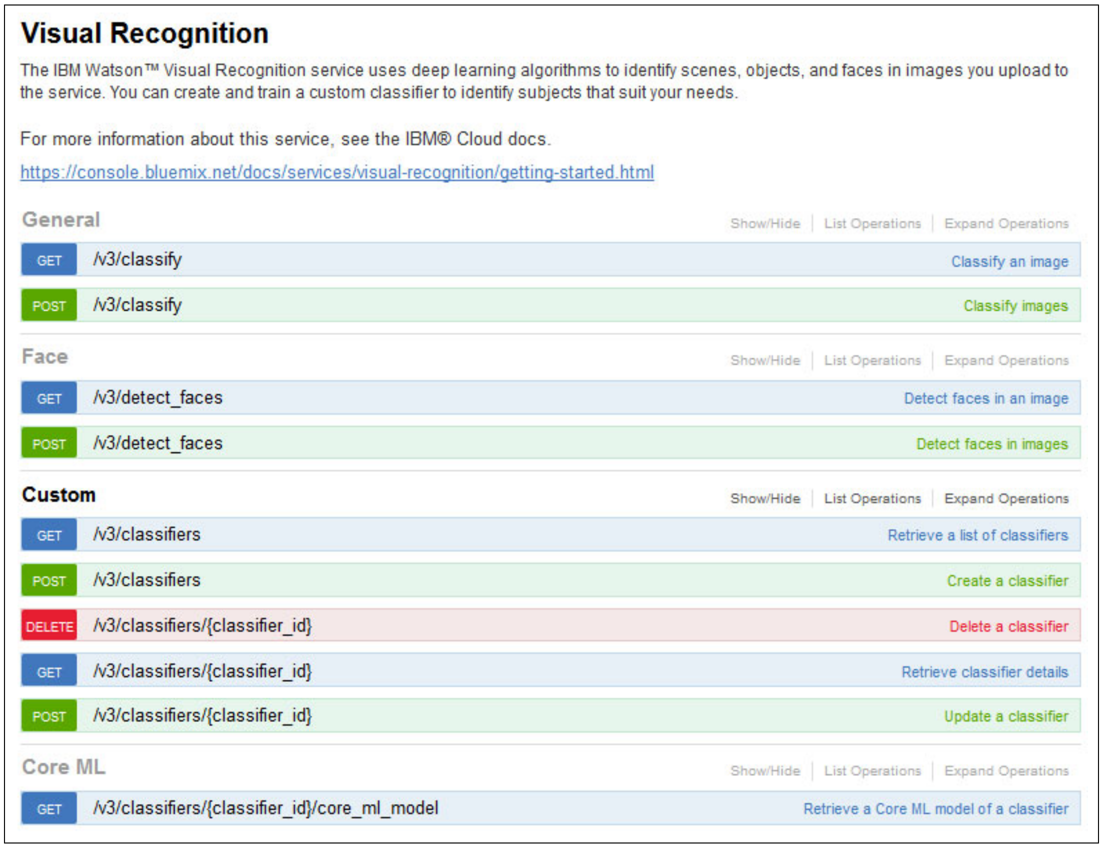
Figure 8-6 Selecting service credentials

6. For security and tracking, you can create multiple API keys. This task can be performed on the Service Credentials window. Use key rotation as explained in the object storage module to reduce the risk of key abuse.

You can also link the API directly to Cloud Foundry applications. This tutorial calls the service through its API for maximum flexibility.

8.4.3 Validation and first use through the API Explorer

Now that you have your API key, you can perform a first picture validation using the API Explorer. Go to the Visual Recognition page for Watson API Explorer, as shown in Figure 8-7.



Visual Recognition

The IBM Watson™ Visual Recognition service uses deep learning algorithms to identify scenes, objects, and faces in images you upload to the service. You can create and train a custom classifier to identify subjects that suit your needs.

For more information about this service, see the IBM® Cloud docs.

https://console.bluemix.net/docs/services/visual-recognition/getting-started.html

General   Show/Hide | List Operations | Expand Operations

GET	/v3/classify	Classify an image
POST	/v3/classify	Classify images

Face   Show/Hide | List Operations | Expand Operations

GET	/v3/detect_faces	Detect faces in an image
POST	/v3/detect_faces	Detect faces in images

Custom   Show/Hide | List Operations | Expand Operations

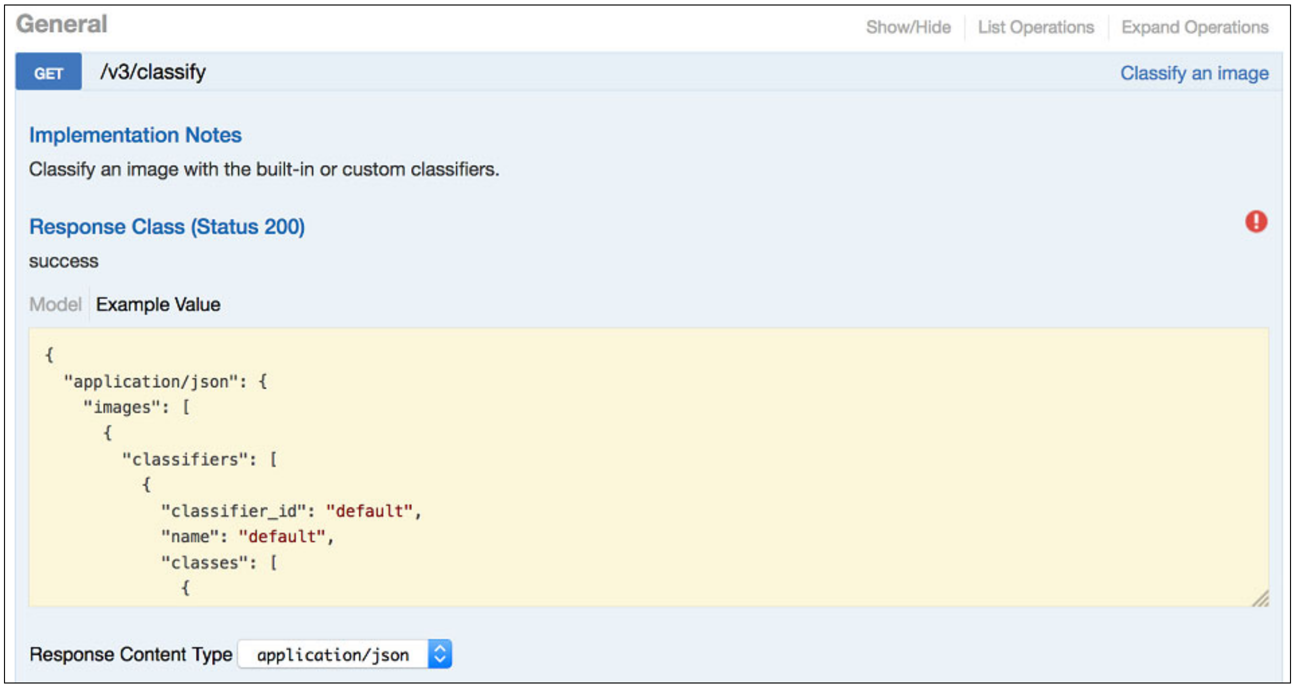
GET	/v3/classifiers	Retrieve a list of classifiers
POST	/v3/classifiers	Create a classifier
DELETE	/v3/classifiers/{classifier_id}	Delete a classifier
GET	/v3/classifiers/{classifier_id}	Retrieve classifier details
POST	/v3/classifiers/{classifier_id}	Update a classifier

Core ML   Show/Hide | List Operations | Expand Operations

GET	/v3/classifiers/{classifier_id}/core_ml_model	Retrieve a Core ML model of a classifier

Figure 8-7 Watson API Explorer

Upcoming sessions cover the different capabilities, so perform a quick validation of your API key by selecting the `<General> - <GET> - </v3/classify>` option, as shown in Figure 8-8.



Implementation Notes

Classify an image with the built-in or custom classifiers.

Response Class (Status 200)

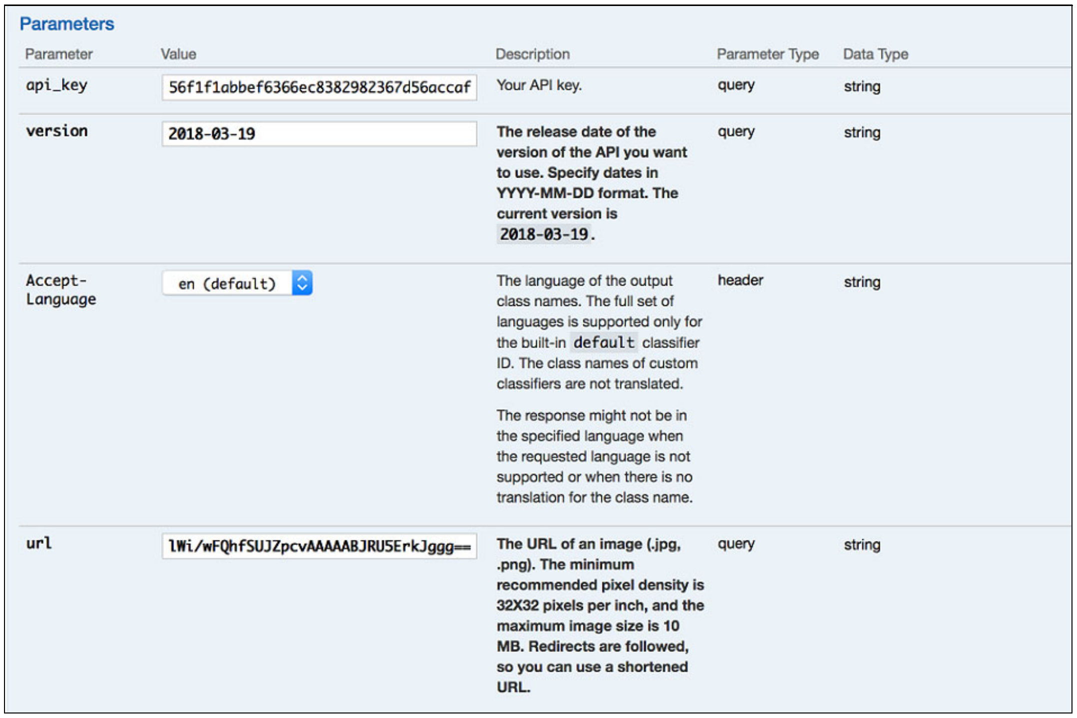
success

```
{
  "application/json": {
    "images": [
      {
        "classifiers": [
          {
            "classifier_id": "default",
            "name": "default",
            "classes": [
              {
```

Response Content Type application/json

Figure 8-8 Using the /v3/classify option

For a simple test, add your API key and a URL to an image. This example uses a well-known IBM image, which is shown in Figure 8-9.

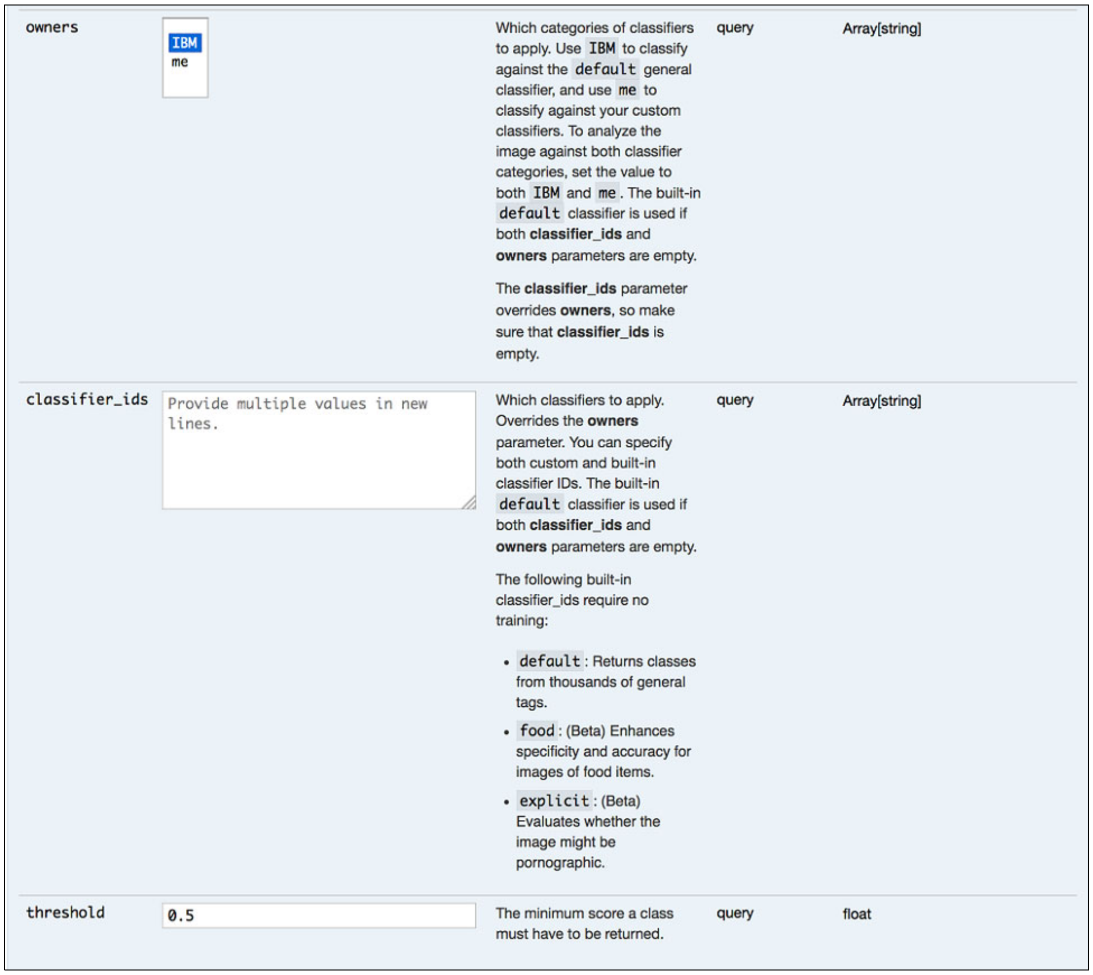


Parameters

Parameter	Value	Description	Parameter Type	Data Type
api_key	56f1f1abbef6366ec8382982367d56accaf	Your API key.	query	string
version	2018-03-19	The release date of the version of the API you want to use. Specify dates in YYYY-MM-DD format. The current version is `2018-03-19`.	query	string
Accept-Language	en (default)	The language of the output class names. The full set of languages is supported only for the built-in `default` classifier ID. The class names of custom classifiers are not translated. The response might not be in the specified language when the requested language is not supported or when there is no translation for the class name.	header	string
url	1Wi/wFQhfSUJZpcvAAAAABJRU5ErkJggg==	The URL of an image (.jpg, .png). The minimum recommended pixel density is 32X32 pixels per inch, and the maximum image size is 10 MB. Redirects are followed, so you can use a shortened URL.	query	string

Figure 8-9 Results of sample image

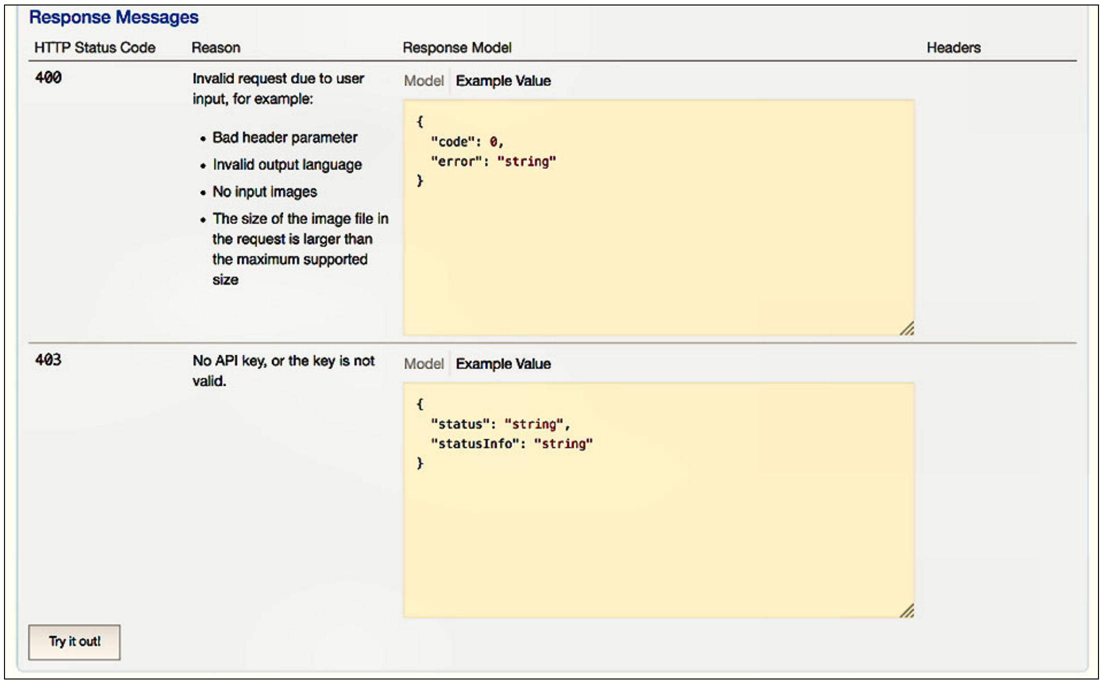
Because we are going to use the default classifier for this first validation, there is no need to focus on the remaining information. These topics are handled in the upcoming sections of this module. Select **IBM** as the owner so that the default classifier is selected, as shown in Figure 8-10. IBM takes data privacy very seriously, so your classifiers will not be shared with any other account.



`owners`	IBM me	Which categories of classifiers to apply. Use `IBM` to classify against the `default` general classifier, and use `me` to classify against your custom classifiers. To analyze the image against both classifier categories, set the value to both `IBM` and `me`. The built-in `default` classifier is used if both **classifier_ids** and **owners** parameters are empty. The **classifier_ids** parameter overrides **owners**, so make sure that **classifier_ids** is empty.	query	Array[string]
`classifier_ids`	Provide multiple values in new lines.	Which classifiers to apply. Overrides the **owners** parameter. You can specify both custom and built-in classifier IDs. The built-in `default` classifier is used if both **classifier_ids** and **owners** parameters are empty. The following built-in classifier_ids require no training: • `default`: Returns classes from thousands of general tags. • `food`: (Beta) Enhances specificity and accuracy for images of food items. • `explicit`: (Beta) Evaluates whether the image might be pornographic.	query	Array[string]
`threshold`	0.5	The minimum score a class must have to be returned.	query	float

Figure 8-10 Selecting the owner

The threshold can be changed if you want to see more output on the same image. The default setting is **0.5**. Click **Try it out** to review your results, as shown in Figure 8-11.


Response Messages

HTTP Status Code	Reason	Response Model	Headers
400	Invalid request due to user input, for example: • Bad header parameter • Invalid output language • No input images • The size of the image file in the request is larger than the maximum supported size	Model Example Value `{ "code": 0, "error": "string" }`	
403	No API key, or the key is not valid.	Model Example Value `{ "status": "string", "statusInfo": "string" }`	

Figure 8-11 Changing the threshold

Example 8-1 shows the results of the example query, which are by default presented in JSON format for easy application usage.

Example 8-1 Results of the example query

```
{
  "images": [
    {
      "classifiers": [
        {
          "classifier_id": "default",
          "name": "default",
          "classes": [
            {
              "class": "Labrador retriever dog",
              "score": 0.831,
              "type_hierarchy": "/animal/domestic animal/dog/retriever
dog/Labrador retriever dog"
            },
            {
              "class": "retriever dog",
              "score": 0.848
            },
            {
              "class": "dog",
              "score": 0.896
            },
            {
              "class": "domestic animal",
              "score": 0.933
            },
            {
              "class": "animal",
              "score": 0.939
            },
            {
              "class": "head of animal",
              "score": 0.59,
              "type_hierarchy": "/animal/domestic animal/head of animal"
            },
            {
              "class": "vizsla dog",
              "score": 0.5,
              "type_hierarchy": "/animal/domestic animal/dog/vizsla dog"
            },
            {
              "class": "light brown color",
              "score": 0.928
            },
            {
              "class": "beige color",
              "score": 0.721
            }
          ]
        }
      ],
      "source_url": "https://pbs.twimg.com/media/DG77KxYUQAAU8OE.jpg",
      "resolved_url": "https://pbs.twimg.com/media/DG77KxYUQAAU8OE.jpg"
    }
  ],
  "images_processed": 1,
  "custom_classes": 0
}
```

By using the default classifier, you already receive a huge amount of information about the image without having to review the image itself. All results shown have more than 50% probability that the image is of a dog of the breed Labrador with a light brown or beige coat color. Compare these results to Figure 8-12.

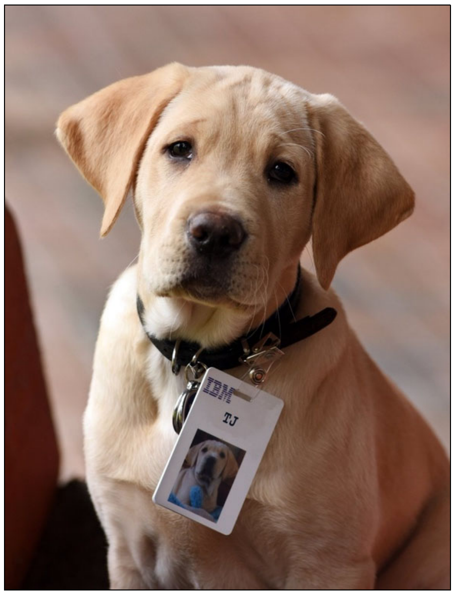


Figure 8-12 Image used in the test

You have now used the Watson Visual Recognition service for the first time and validated that your API key works as expected. Read more about this great Watson technology story at Passion//Project.

8.4.4 Implementation

Using IBM Watson Visual Recognition through the API provides a solid and easy implementation. For code samples for different languages that can help you to add this service to many other applications, see the IBM Watson Visual Recognition wiki.

API requests require a version parameter that takes a date in the format `version=YYYY-MM-DD`. When you change the API in a backwards-incompatible way, a new version date is associated with (or given to) the API. This feature ensures that the same result is obtained when using the API at the same timestamp and that future enhancements of the API will not result in a different output on an already used image.

Send the version parameter with every API request. The service uses the API version for the date you specify, or the most recent version before that date. Do not default to the current date. Instead, specify a date that matches a version that is compatible with your app, and do not change it until your app is ready for a later version.

Tip: This documentation describes the current version of Watson Visual Recognition, 2018-03-19. In some cases, differences in earlier versions are noted in the descriptions of parameters and response models.

8.4.5 Using Watson Visual Recognition Service with the default classifier

To use the Watson Visual Recognition Service with the default classifier, complete these steps:

1. Call the visual recognition service within our application. We also require Lodash, which is a powerful tool for parsing the gathered JSON data. Line 4 creates the call method for the VR service with current version and `api_key`, as shown in Example 8-2.

Example 8-2 Adding Lodash

```
// Add the watson-developer-cloud toolbox to allow accessing the Watson Visual
Recognition integration
var VisualRecognitionV3 =
require("watson-developer-cloud/visual-recognition/v3");
// Add lodash which is used for data filtering and parsing
var _ = require("lodash");
// Define a VisualRecognition object with the current version and your API key
var visualRecognition = new VisualRecognitionV3({
   version: "2018-03-19",
   api_key: "fb080f8e3f8f54177bfbbf7987b424f044241bd8"
});
```

2. Export your function to the rest of the application so it can be easily called in the other modules, as shown in Example 8-3.

Example 8-3 Exporting the function

```
// Create Public methods to call from the main application
module.exports.getCarColor = _get_car_color;
module.exports.getCarPosition = _get_car_position;
module.exports.getCarDamage = _get_car_damage;
module.exports.validateCar = _validate_car;
```

3. Use your Watson Visual Recognition service to validate the contents of an image and take appropriate actions in the front end, as shown in Example 8-4.

Example 8-4 Validating the contents of an image

```
/*
The _validate_car function is used to validate if the uploaded picture has a
car recognized.
A smart protection to validate the end-user input and return an error when they
have selected the wrong image.

This function uses the default watson visual recognition classifier and a
threshold at 0 to allow low detection rate to still pass this check.
After filtering the result with lodash we receive an array with all the items
that have "car" within their classname.
As last we sort the array with highest probability first.
*/

function _validate_car(image) {
   return new Promise(function(resolve, reject) {

      var car_classifier_ids = ["default"];
      var car_threshold = 0;
      var params_carcol = {
         images_file: image,
         classifier_ids: car_classifier_ids,
         threshold: car_threshold
      };

      visualRecognition.classify(params_carcol, function(err, response) {
         if (err) {
            reject(err);
         }
         else {
            var car_validation = 
_.filter(response.images[0].classifiers[0].classes, function(filter_car) {
               return filter_car.class.includes("car");
            });
            var car_validation_sorted = _.orderBy(car_validation, ["score", 
"class"], ["desc", "asc"]);
            resolve(car_validation_sorted);
         }
      });
   });
}
```

4. The second function is also for validation purposes and uses the same default classifier. It reviews the colors that are detected within the image and lists them in a descending array with the highest probability first. This step is used to validate whether the car color of the registered vehicle matches the provided pictures, as shown in Example 8-5.

Example 8-5 Validating the car color

```
/*
The _get_car_color function is to obtain the car color which than can be 
matched with the contract car colour as validation.
A smart protection to validate the end-user input and return an error when they 
have selected a non matching car with their contract.
Be carefull since the default class has a huge range of colours which do not 
always match classic colours (f.e. Teal is a variation of blue),
So when implementing this part it would be a best practices to review the color 
results of a new registered car and add them to the filter.
Another enhancement would be to crop the car from the image, since now a 
zoom-out picture with trees could provide green as most dominant colour.

This function uses the default watson visual recognition classifier and a 
threshold at 0 to allow low detection rate to still pass this check.
After filtering the result with lodash we receive an array with all the items 
that have "color" within their classname.
As last we sort the array with highest probability first.
*/

function _get_car_color(image) {
   return new Promise(function(resolve, reject) {

      var carcol_classifier_ids = ["default"];
      var carcol_threshold = 0;

      var params_carcol = {
            images_file: image,
            classifier_ids: carcol_classifier_ids,
            threshold: carcol_threshold
         };

         visualRecognition.classify(params_carcol, function(err, response) {
            if (err) {
               reject(err);
            }
            else {
               var car_color = _.filter(response.images[0].classifiers[0].classes,
function(filter_color) {
                  return filter_color.class.includes("color");
               });
               var car_color_sort = _.orderBy(car_color, ["score", "class"],
["desc", "asc"]);
               resolve(car_color_sort);
            }
         });
      });
}
```

8.4.6 Using Watson VR Service with custom classifiers with a single class

To be able to continue with the next parts of the code, you must create the first custom classifier. The first classifier requires a paid subscription because it requires four distinct classifiers to work. At the time of writing this IBM Redpaper, a lite account only allows a single custom classifier.

The reasoning behind a custom classifier is that you can have multiple positive classes and a single negative class. Because the first custom classifier tries to automatically detect the left, right, front, and back of a car, a single positive class and three negative classes are put in a single image pool to correspond to a single negative class.

With the downloaded examples, your classifiers will look like this where the positive class will have 11 pictures from cars that match the angle and the negatives class will have 33 pictures of angles that not match this classifier (11 per angle).

```
Car_right_classifier = +(car_right_small_positive_examples)
-(car_right_small_negatives)
Car_left_classifier = +(car_left_small_positive_examples)
-(car_left_small_negatives)
Car_front_classifier = +(car_front_small_positive_examples)
-(car_front_small_negatives)
Car_back_classifier = +(car_back_small_positive_examples)
-(car_back_small_negatives)
```

Now train your Watson Visual Recognition service using these images. You can perform this training using the demonstrated API Explorer that you used to validate your API key. Use **`curl`** to send the data to the VR engine.

For every classifier, issue the command shown in Example 8-6 from the 1.1 directory that you downloaded.

Example 8-6 The curl command

```
# First we define our curl command and add the corresponding headers for
succesfull exectution.
# The first input is a single positive classifier in this example.
"classname_positive_examples" defines a classname that has to be put into the
positive examples side.
# There is no class name available for negative examples since they are not used
within a response. Like with the positive examples it is important to have
sufficient variation.
# Last custom attribute is the name of the classifier, a custom number will be
added to remove the risk of having a duplicate name within the VR service.
# Add the api-explorer link to the end including your active API key and version
information.

curl -X POST --header 'Content-Type: multipart/form-data' --header 'Accept:
application/json'
-F "car_angle_positive_examples=@car_angle_positive_examples.zip"
-F "negative_examples=@car_angles_negatives.zip"
-F name = 'custom-classifier-name'
'https://watson-api-explorer.mybluemix.net/visual-recognition/api/v3/classifiers?a
pi_key=api-key-here&version=2018-03-19'
```

Repeat this **curl** command three more times for the remaining angles. The second iteration fails if you do not have a paid subscription because only one custom classifier is allowed in the Lite plan.

The response shown in Example 8-7 is expected for each classifier.

Example 8-7 Expected results of the curl command

```
{
    "classifier_id": "cosredbook_car_right_213832081",
    "name": "cosredbook_car_right",
    "status": "training",
    "owner": "10ee4bc4-bc5e-4a55-8258-740998a8b5e5",
    "created": "2018-05-08T10:41:24.350Z",
    "updated": "2018-05-08T10:41:24.350Z",
    "classes": [
        {
            "class": "car_right"
        }
    ],
    "core_ml_enabled": true
}
```

Now you must wait for the training to complete to be able to use your custom classifiers. Because this example uses 11 images per class, the training only takes a few minutes, after which the custom classifier will be available when using the same API key. To check the training status, use the API Explorer or the **curl** command shown in Example 8-8.

Example 8-8 Using the curl command for training

```
curl "https://gateway-a.watsonplatform.net/visual-recognition/api/v3/classifiers/
      custom-classifier-name?
      api_key={api-key}&
      version=2018-03-19"
```

We received a “ready” status after about five minutes. After this time, you can use your customer created classifier within your code, as shown in Example 8-9.

Example 8-9 Using the customer created classifier

```
{
    "classifier_id": "cosredbook_car_right_213832081",
    "name": "cosredbook_car_right",
    "status": "ready",
    "owner": "10ee4bc4-bc5e-4a55-8258-740998a8b5e5",
    "created": "2018-05-08T10:41:24.350Z",
    "updated": "2018-05-08T10:41:24.350Z",
    "classes": [
        {
            "class": "car_right"
        }
    ],
    "core_ml_enabled": true
}
```

Now test your classifier on one of the positive pictures to see whether it is functioning as expected. Example 8-10 uses the **curl** command, but you can use the Watson API Explorer if you prefer a graphical form.

Example 8-10 Testing the classifier on a positive picture

```
curl - X GET--header 'Accept: application/json' --header 'Accept-Language: en'
  'https://watson-api-explorer.mybluemix.net/visual-recognition/api/v3/classify?
  api_key={api-key}&
  version=2018-03-19&
  url=link-to-image&
  =me&
  classifier_ids=custom-classifier-name&
  threshold=0.5'
```

The **curl** command in Example 8-10 on page 129 reviews the image shown in Figure 8-13 using the custom classifier for right and shows any result that has a probability above 0.5.

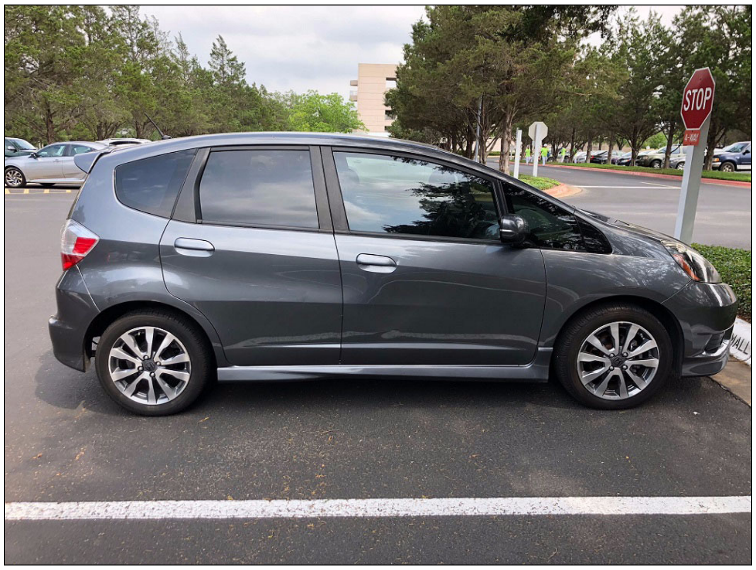


Figure 8-13 CAR_TEST_2_OR_RIGHT.JPG

Example 8-11 shows the result of the API Call.

Example 8-11 Result of the positive test

```
{
    "images": [
        {
          "classifier_id": "cosredbook_car_right_213832081",
          "name": "cosredbook_car_right",
          "classes": [
            {
              "class": "car_right",
              "score": 0.697
            }
                    ]
                }
],
      "source_url": 
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_2_OR_RIGHT.
JPG",
      "resolved_url": 
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_2_OR_RIGHT.
JPG"
    }
    ],
    "images_processed": 1,
    "custom_classes": 1
}
```

The custom classifier is 69.7% sure that it is the right side of the car. Now review the same car but the left side as shown in Figure 8-14 to see what probability it receives for the right side classifier.

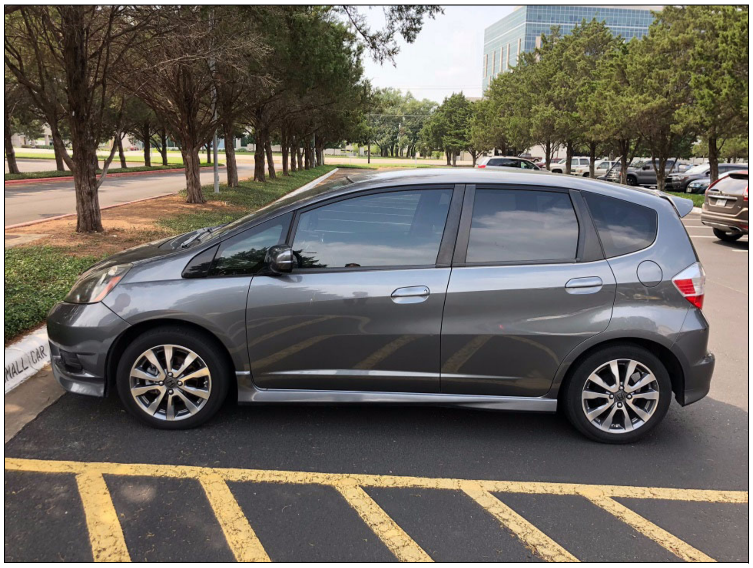


Figure 8-14 CAR_TEST_2_OR_LEFT.JPG

Example 8-12 shows the code involved.

Example 8-12 Code to classify the right side using a left side image

```
{
  "images": [
    {
      "classifiers": [
        {
          "classifier_id": "cosredbook_car_right_213832081",
          "name": "cosredbook_car_right",
          "classes": [
            {
              "class": "car_right",
              "score": 0.207
            }
          ]
        }
      ],
      "source_url":
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_2_OR_LEFT.J
PG",
      "resolved_url":
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_2_OR_LEFT.J
PG"
    }
  ],
  "images_processed": 1,
  "custom_classes": 1
}
```

It is clear that when comparing both, it will be a valid classification. However, you need to add other classifiers to be able to validate the real angle of this image within one API call.

First, review whether all your classifiers are ready to be used. Issue the **curl** command shown in Example 8-13 to perform the validation.

Example 8-13 Performing the validation

```
curl - X GET--header 'Accept: application/json'

'https://watson-api-explorer.mybluemix.net/visual-recognition/api/v3/classifiers?
  api_key={api-key}&
  version=2018-03-19'
```

You should get the output shown in Example 8-14.

Example 8-14 Output from the validation command

```
{
  "classifiers": [
    {
      "classifier_id": "cosredbook_car_front_1204165740",
      "name": "cosredbook_car_front",
      "status": "ready"
    },
    {
      "classifier_id": "cosredbook_car_left_1405197292",
      "name": "cosredbook_car_left",
      "status": "ready"
    },
    {
      "classifier_id": "cosredbook_car_right_213832081",
      "name": "cosredbook_car_right",
      "status": "ready"
    },
    {
      "classifier_id": "cosredbook_car_back_974985661",
      "name": "cosredbook_car_back",
      "status": "ready"
    },
  ]
}
```

When all four classifiers are ready, you can run the request by evaluating all four angles at once instead of one-by-one. This process makes it easier to compare the angle probabilities and decide which angle is the most likely, as shown in Example 8-15.

Example 8-15 Comparing the angle probabilities

```
{
  "images": [
    {
      "classifiers": [
        {
          "classifier_id": "cosredbook_car_right_213832081",
          "name": "cosredbook_car_right",
          "classes": [
            {
              "class": "car_right",
              "score": 0.207
            }
          ]
        },
        {
          "classifier_id": "cosredbook_car_back_974985661",
          "name": "cosredbook_car_back",
          "classes": [
            {
              "class": "car_back",
              "score": 0.001
            }
          ]
        },
        {
          "classifier_id": "cosredbook_car_front_1204165740",
          "name": "cosredbook_car_front",
          "classes": [
            {
              "class": "car_front",
              "score": 0.019
            }
          ]
        },
        {
          "classifier_id": "cosredbook_car_left_1405197292",
          "name": "cosredbook_car_left",
          "classes": [
            {
              "class": "car_left",
              "score": 0.485
            }
          ]
        }
      ],
      "source_url":
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_2_OR_LEFT.J
PG",
      "resolved_url":
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_2_OR_LEFT.J
PG"
    }
  ],
  "images_processed": 1,
  "custom_classes": 4
}
```

The output in Example 8-15 on page 132 gives a solid left result when compared to the other classes, even though the overall probability is below 50%. Note that this data set is in its early stages, so you must further enhance the data classification.

Now implement these four custom classifiers within your `node.js` application, as shown in Example 8-16.

Example 8-16 Implementing the custom classifiers

```
/*
The _get_car_position is our self-trained class which we will update frequently
throughout the tutorial. This class requires a paid subscription since we will be
using 4 custom classes.
Make sure to first follow the training steps before enabling this feature within
your code. Update the classifier ids to make it work.

This function uses our four self-trained classifiers and a threshold at 0 to allow
low detection rate to still be added to the array.
We do not need to filter the results since there are only 4 classes, 1 per angle
and we want to keep all four withing the returned array.
We sort the array with highest probability first where the first element of the
array [0] will be used as automatic angle.
*/

function _get_car_position(image) {
   return new Promise(function(resolve, reject) {

      var carpos_classifier_ids = ["cosredbook_car_right_small3_39505551",
"cosredbook_car_left_small5_1133497139", "cosredbook_car_front_small_740382583",
"cosredbook_car_back_1854234732"];
      var carpos_threshold = 0;

      var params_carpos = {
         images_file: image,
         classifier_ids: carpos_classifier_ids,
         threshold: carpos_threshold
      };

      visualRecognition.classify(params_carpos, function(err, response) {
         if (err)
            reject(err);
         else
            var car_position = _.orderBy(response.images[0].classifiers,
["classes[0].score", "classes[0].class"], ["desc", "asc"]);
         resolve(car_position);
      });
   });
}
```

8.4.7 Using Watson VR Service with a multi-class custom classifier

You can use the Watson VR Service with a multi-class classifier either by using the API or with Watson Studio.

Using the API

In this example, you create a single classifier with multiple classes that performs car damage detection. This use-case has multiple positive classes (flat tire, broken windshield, broken mirror, car accident, and so on) versus a single negative class (car without damage). The pictures used in this example are gathered from various sources. To perform this part, you must first obtain at least 10 images per positive class to have a valid class. Make sure that you have sufficient variation.

We have downloaded some images for every class, each containing around 20 images:

```
car_damage_broken_mirror_positive_examples.zip
car_damage_flat_tire_positive_examples.zip
car_damage_broken_window_positive_examples.zip
car_damage_accident_positive_examples.zip
```

In addition, we include a set of undamaged cars:

```
car_damage_negatives.zip
```

First, add two positive classes to the same qualifier, and then include an extra class to the same qualifier. The same process can be used to add images to an existing class. You will see the difference shown in Example 8-17 in the API call.

Example 8-17 Adding new classes

```
curl - X POST--header 'Content-Type: multipart/form-data' --header 'Accept:
application/json'
-F
"car_damage_broken_window_positive_examples=@car_damage_broken_window_positive_exa
mples.zip"
-F
"car_damage_flat_tire_positive_examples=@car_damage_flat_tire_positive_examples.zi
p"
-F "negative_examples=@car-damage_negatives.zip"
-F name = "cosredbook_car_damage"
'https://watson-api-explorer.mybluemix.net/visual-recognition/api/v3/classifiers?a
pi_key=api-key-here&version=2018-03-19'
```

An example output on the classifier details is shown in Example 8-18.

Example 8-18 Output of the classifier details

```
{
    "classifier_id": "cosredbook_car_damage_1460547759",
    "name": "cosredbook_car_damage",
    "status": "training",
    "owner": "10ee4bc4-bc5e-4a55-8258-740998a8b5e5",
    "created": "2018-05-08T13:40:56.987Z",
    "updated": "2018-05-08T13:40:56.987Z",
    "classes": [
        {
            "class": "car_damage_flat_tire"
        },
        {
            "class": "car_damage_broken_window"
        }
    ],
    "core_ml_enabled": true
}
```

Example 8-19 shows the output of this class using the `direct-api` approach from the image shown in Figure 8-15.

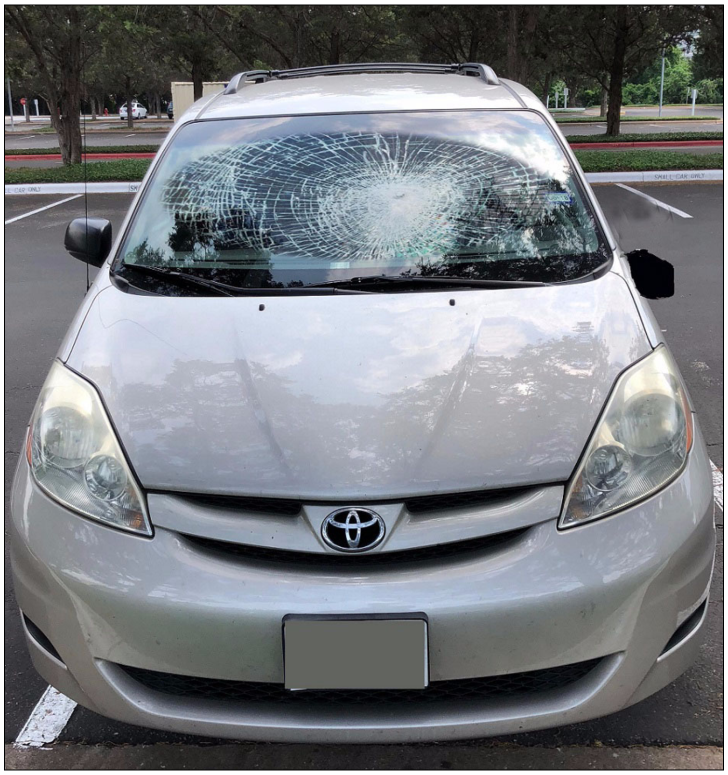


Figure 8-15 CAR_TEST_7_OR_FRONT_broken.jpg

Example 8-19 shows the output of this class.

Example 8-19 Output of the direct-api approach

```
{
    "images": [
        {
            "classifiers": [
                {
                    "classifier_id": "cosredbook_car_damage_1460547759",
                    "name": "cosredbook_car_damage",
                    "classes": [
                        {
                            "class": "car_damage_broken_window",
                            "score": 0.116
                        },
                        {
                            "class": "car_damage_flat_tire",
                            "score": 0.001
                        },
                    ]
                }
            ],
            "source_url":
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_7_OR_FRONT_
broken.jpg",
            "resolved_url":
"https://s3-api.us-geo.objectstorage.softlayer.net/cosredbook/CAR_TEST_7_OR_FRONT_
broken.jpg"
        }
    ],
    "images_processed": 1,
    "custom_classes": 2
}
```

Now you can add another positive class to an existing classifier when training is completed. You can also add more images to one of the existing positive classes or the negative class, as shown in Example 8-20.

Example 8-20 Adding another positive class

```
curl - X POST--header 'Content-Type: multipart/form-data'--header 'Accept:
application/json'
  -F
"car_damage_broken_mirror_positive_examples=@car_damage_broken_mirror_positive_exa
mples.zip"
  -F "negative_examples=@car-damage_negatives.zip"
  -F name="cosredbook_car_damage_1460547759"

'https://watson-api-explorer.mybluemix.net/visual-recognition/api/v3/classifiers?a
pi_key=api-key-here&version=2018-03-19'
```

Now you notice that the status is re-training, which means the model is being redefined. When the status is ready, the class is available again for usage, as shown in Example 8-21.

Example 8-21 Class ready for use

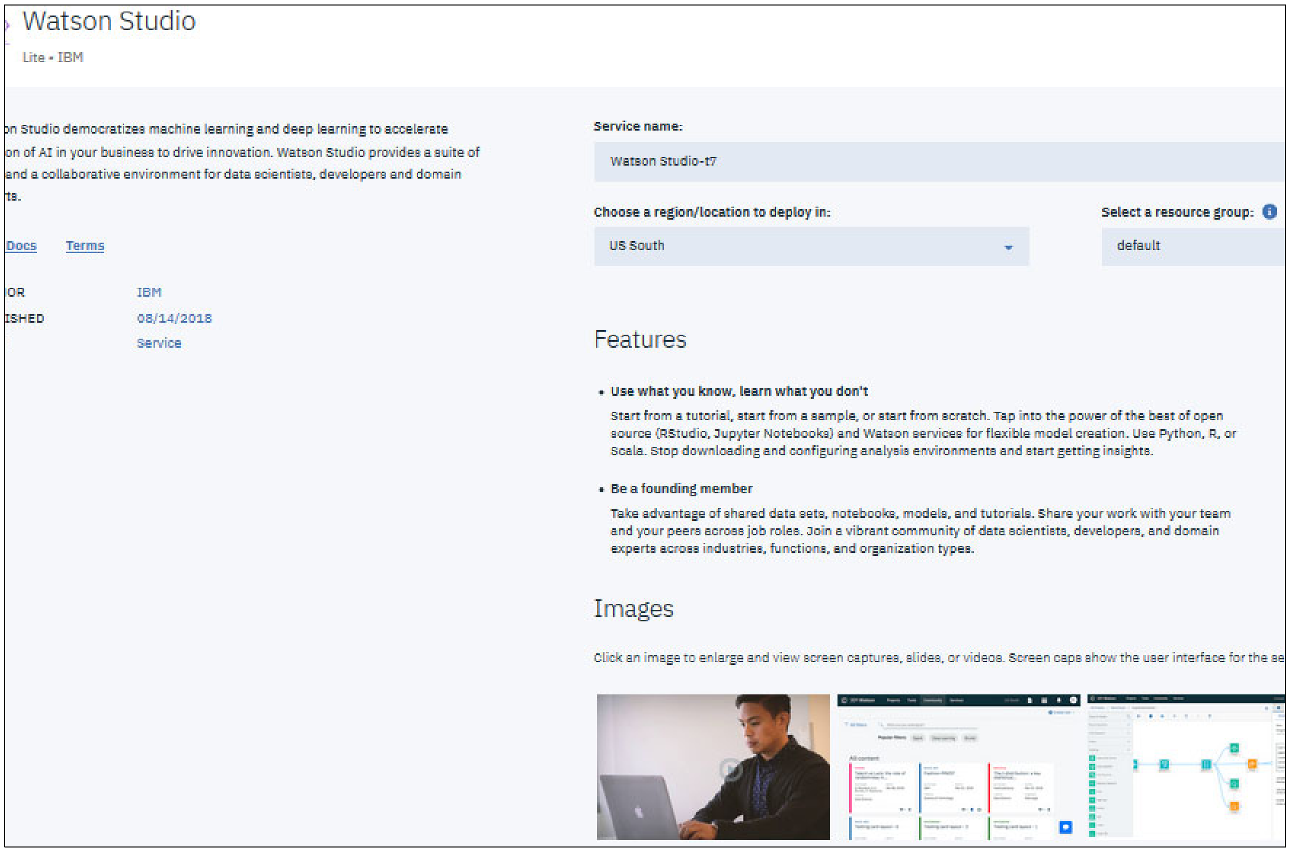
```
{
  "classifier_id": "cosredbook_car_damage_1460547759",
  "name": "cosredbook_car_damage",
  "status": "retraining",
  "owner": "10ee4bc4-bc5e-4a55-8258-740998a8b5e5",
  "created": "2018-05-08T13:40:56.987Z",
  "updated": "2018-05-08T13:40:56.987Z",
  "classes": [
    {
      "class": "car_damage_broken_mirror"
    },
    {
      "class": "car_damage_flat_tire"
    },
    {
      "class": "car_damage_broken_window"
    }
  ],
  "core_ml_enabled": true
}
```

Using Watson Studio

The new Watson Studio provides a centralized console for any Watson AI related task. This section repeats the steps in “Using the API” on page 135 using a graphical user interface. The data is not only available for your Visual Recognition module, but also for all the other Watson modules.

Complete the following steps:

1. Enable the Watson Studio (Figure 8-16), which is available in the catalog, by clicking **AI** and then selecting **Watson Studio**. You can sign up for a Lite (free) plan for this exercise because you are only going to experiment with the VR service in this module.



Watson Studio

Lite • IBM

on Studio democratizes machine learning and deep learning to accelerate on of AI in your business to drive innovation. Watson Studio provides a suite of and a collaborative environment for data scientists, developers and domain ts.

Docs Terms

OR	IBM
ISHED	08/14/2018
	Service

Service name: Watson Studio-t7

Choose a region/location to deploy in: US South

Select a resource group: default

Features

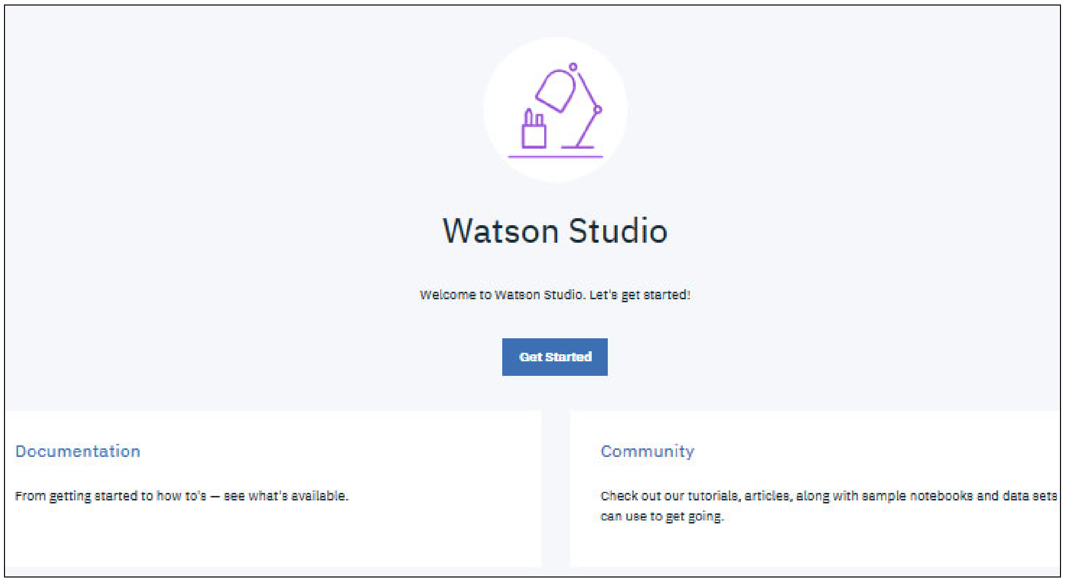
- **Use what you know, learn what you don't**
  Start from a tutorial, start from a sample, or start from scratch. Tap into the power of the best of open source (RStudio, Jupyter Notebooks) and Watson services for flexible model creation. Use Python, R, or Scala. Stop downloading and configuring analysis environments and start getting insights.
- **Be a founding member**
  Take advantage of shared data sets, notebooks, models, and tutorials. Share your work with your team and your peers across job roles. Join a vibrant community of data scientists, developers, and domain experts across industries, functions, and organization types.

Images

Click an image to enlarge and view screen captures, slides, or videos. Screen caps show the user interface for the se

Figure 8-16 Watson Studio

2. Click **Get Started** to open the dashboard shown in Figure 8-17.



Watson Studio

Welcome to Watson Studio. Let's get started!

Documentation

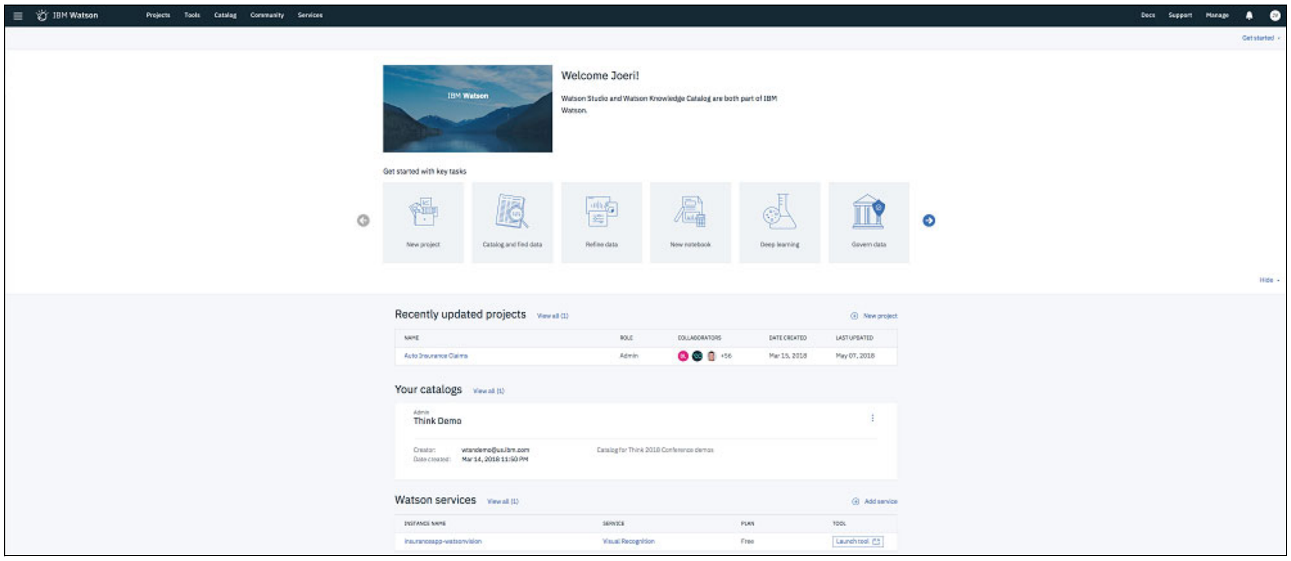
From getting started to how to's — see what's available.

Community

Check out our tutorials, articles, along with sample notebooks and data sets
can use to get going.

Figure 8-17 Watson Studio dashboard

3. The Watson Studio dashboard displays all of the available services, as shown in Figure 8-18. The dashboard offers a centralized storage repository on object storage where all services are automatically linked. You can create a project that creates a space where you can work with many collaborators. You can share the services on an individual or project basis with your co-workers.



Welcome Joeri!

Watson Studio and Watson Knowledge Catalog are both part of IBM Watson.

Get started with key tasks

Recently updated projects View all (1) New project

NAME	ROLE	COLLABORATORS	DATE CREATED	LAST UPDATED
Auto Insurance Claims	Admin	+56	Mar 15, 2018	May 07, 2018

Your catalogs View all (1)

Admin
Think Demo

Creator: wtandemo@us.ibm.com
Date created: Mar 14, 2018 11:50 PM

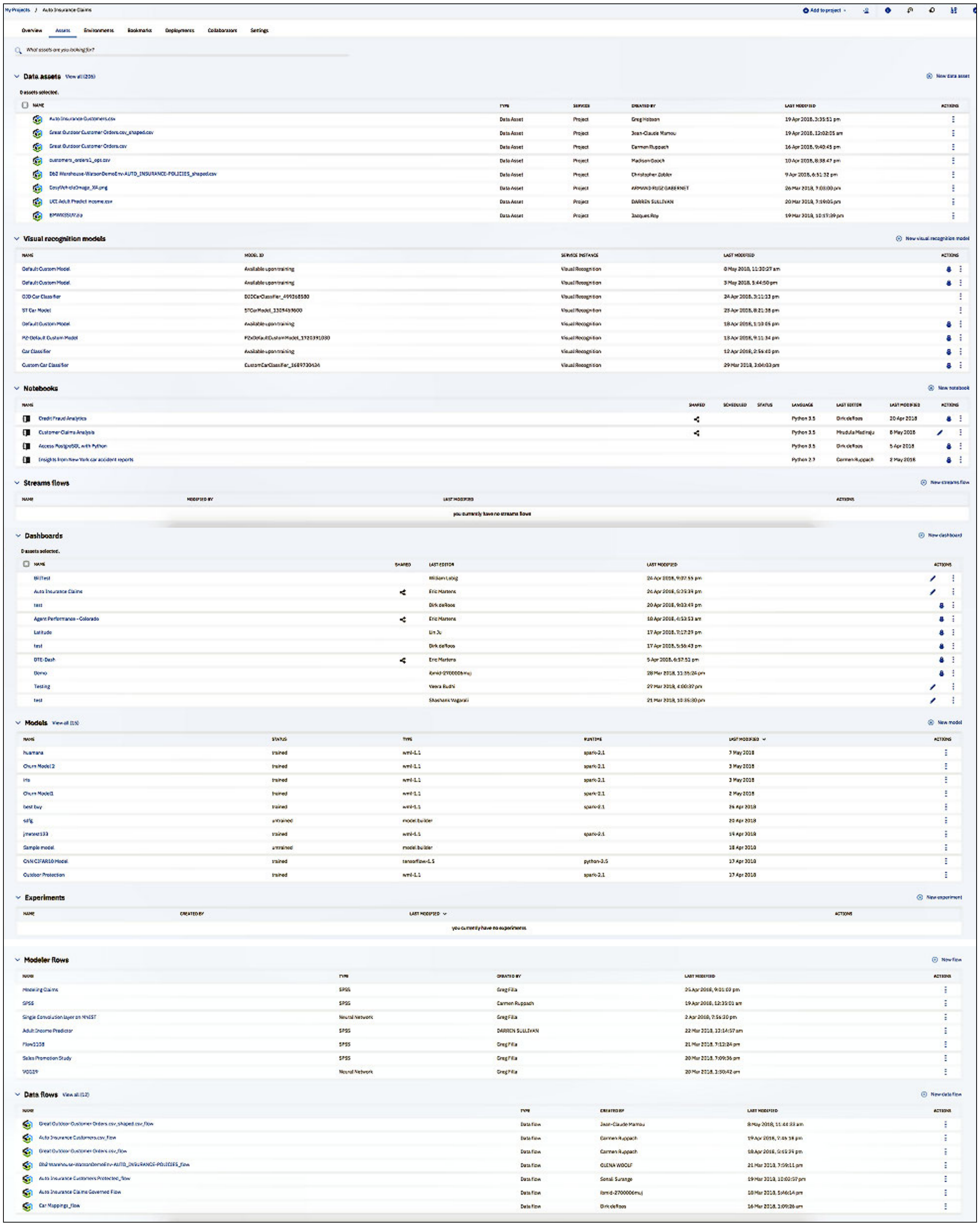
Catalog for Think 2018 Conference demos

Watson services View all (1) Add service

INSTANCE NAME	SERVICE	PLAN	TOOL
insuranceapp-watsonvision	Visual Recognition	Free	Launch tool

Figure 8-18 Watson Studio dashboard showing available services

4. Open the Auto Insurance Claims project to show an overview of all the related services, as shown in Figure 8-19.


My Projects / Auto Insurance Claims

Data assets View all (206)

0 assets selected.

NAME	TYPE	SERVICE	CREATED BY	LAST MODIFIED	ACTIONS
Auto Insurance Customers.csv	Data Asset	Project	Greg Hobson	19 Apr 2018, 3:35:51 pm	⋮
Great Outdoor Customer Orders.csv_shaped.csv	Data Asset	Project	Jean-Claude Mamou	19 Apr 2018, 12:02:05 am	⋮
Great Outdoor Customer Orders.csv	Data Asset	Project	Carmen Ruppach	16 Apr 2018, 9:40:45 pm	⋮
customers_orders1_opt.csv	Data Asset	Project	Madison Gooch	10 Apr 2018, 8:38:47 pm	⋮
Db2 Warehouse-WatsonDemoEnv-AUTO_INSURANCE-POLICIES_shaped.csv	Data Asset	Project	Christopher Zobler	9 Apr 2018, 6:51:32 pm	⋮
CopyVehicleImage_XA.png	Data Asset	Project	ARMAND RUIZ GABERNET	26 Mar 2018, 7:03:00 pm	⋮
UCI Adult Predict income.csv	Data Asset	Project	DARREN SULLIVAN	20 Mar 2018, 7:19:05 pm	⋮
BMWX5SUV.zip	Data Asset	Project	Jacques Roy	19 Mar 2018, 10:17:39 pm	⋮

Visual recognition models

NAME	MODEL ID	SERVICE INSTANCE	LAST MODIFIED	ACTIONS
Default Custom Model	Available upon training	Visual Recognition	8 May 2018, 11:30:27 am	🔒 ⋮
Default Custom Model	Available upon training	Visual Recognition	3 May 2018, 5:44:50 pm	🔒 ⋮
DJD Car Classifier	DJDCarClassifier_499368580	Visual Recognition	24 Apr 2018, 3:11:33 pm	⋮
ST Car Model	STCarModel_1309459600	Visual Recognition	25 Apr 2018, 8:21:38 pm	⋮
Default Custom Model	Available upon training	Visual Recognition	18 Apr 2018, 1:10:06 pm	🔒 ⋮
PZ-Default Custom Model	PZxDefaultCustomModel_1720391080	Visual Recognition	13 Apr 2018, 9:11:34 pm	🔒 ⋮
Car Classifier	Available upon training	Visual Recognition	12 Apr 2018, 2:56:40 pm	🔒 ⋮
Custom Car Classifier	CustomCarClassifier_1689730424	Visual Recognition	29 Mar 2018, 3:04:03 pm	🔒 ⋮

Notebooks

NAME	SHARED	SCHEDULED	STATUS	LANGUAGE	LAST EDITOR	LAST MODIFIED	ACTIONS
Credit Fraud Analytics	✓			Python 3.5	Dirk deRoos	20 Apr 2018	🔒 ⋮
Customer Claims Analysis	✓			Python 3.5	Mrudula Madiraju	8 May 2018	✎ ⋮
Access PostgreSQL with Python				Python 3.5	Dirk deRoos	5 Apr 2018	🔒 ⋮
Insights from New York car accident reports				Python 2.7	Carmen Ruppach	2 May 2018	🔒 ⋮

Streams flows

NAME	MODIFIED BY	LAST MODIFIED	ACTIONS
		you currently have no streams flows	

Dashboards

0 assets selected.

NAME	SHARED	LAST EDITOR	LAST MODIFIED	ACTIONS
BillTest		William Lobig	24 Apr 2018, 9:07:55 pm	✎ ⋮
Auto Insurance Claims	✓	Eric Martens	26 Apr 2018, 6:29:34 pm	✎ ⋮
test		Dirk deRoos	20 Apr 2018, 9:03:49 pm	🔒 ⋮
Agent Performance - Colorado	✓	Eric Martens	18 Apr 2018, 4:53:53 am	🔒 ⋮
Latitude		Lin Ju	17 Apr 2018, 7:17:29 pm	🔒 ⋮
test		Dirk deRoos	17 Apr 2018, 5:56:43 pm	🔒 ⋮
DTE-Dash	✓	Eric Martens	5 Apr 2018, 6:57:53 pm	🔒 ⋮
Demo		ibmid-270000bmuj	28 Mar 2018, 11:35:24 pm	🔒 ⋮
Testing		Veera Budhi	27 Mar 2018, 4:00:37 pm	✎ ⋮
test		Shashank Vagarali	21 Mar 2018, 10:36:30 pm	✎ ⋮

Models View all (15)

NAME	STATUS	TYPE	RUNTIME	LAST MODIFIED	ACTIONS
humana	trained	wml-1.1	spark-2.1	7 May 2018	⋮
Churn Model 2	trained	wml-1.1	spark-2.1	3 May 2018	⋮
iris	trained	wml-1.1	spark-2.1	3 May 2018	⋮
Churn Model1	trained	wml-1.1	spark-2.1	2 May 2018	⋮
best buy	trained	wml-1.1	spark-2.1	26 Apr 2018	⋮
sdfg	untrained	model builder		20 Apr 2018	⋮
jmeroer133	trained	wml-1.1	spark-2.1	19 Apr 2018	⋮
Sample model	untrained	model builder		18 Apr 2018	⋮
CNN CIFAR10 Model	trained	tensorflow-1.5	python-3.5	17 Apr 2018	⋮
Outdoor Protection	trained	wml-1.1	spark-2.1	17 Apr 2018	⋮

Experiments

NAME	CREATED BY	LAST MODIFIED	ACTIONS
		you currently have no experiments	

Modeler flows

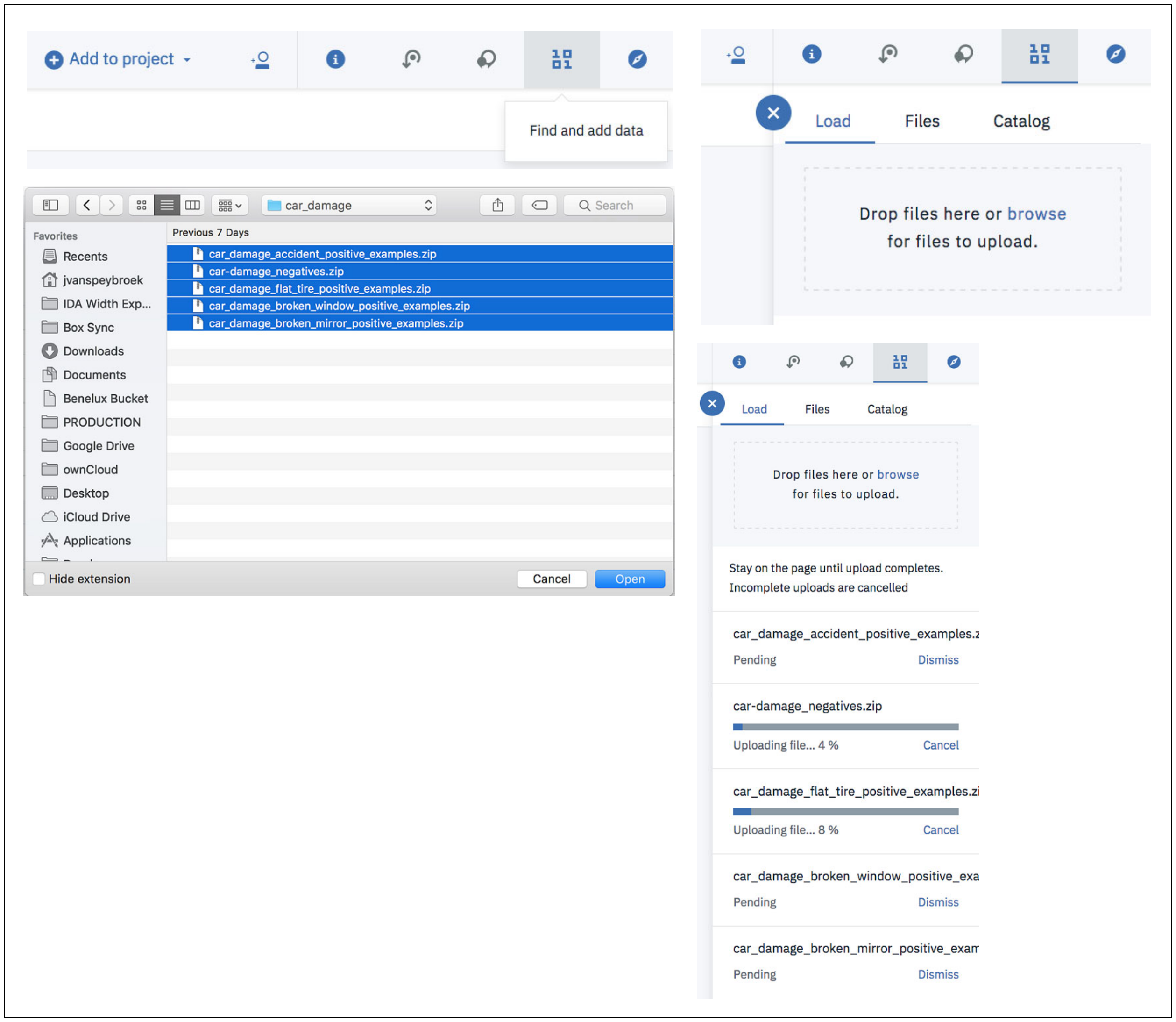
NAME	TYPE	CREATED BY	LAST MODIFIED	ACTIONS
Modeling Claims	SPSS	Greg Filla	25 Apr 2018, 9:01:03 pm	⋮
SPSS	SPSS	Carmen Ruppach	19 Apr 2018, 12:35:01 am	⋮
Single Convolution layer on MNIST	Neural Network	Greg Filla	2 Apr 2018, 7:56:10 pm	⋮
Adult Income Predictor	SPSS	DARREN SULLIVAN	22 Mar 2018, 11:14:57 am	⋮
Flow1108	SPSS	Greg Filla	21 Mar 2018, 7:12:24 pm	⋮
Sales Promotion Study	SPSS	Greg Filla	20 Mar 2018, 7:09:36 pm	⋮
VGG19	Neural Network	Greg Filla	20 Mar 2018, 3:50:42 am	⋮

Data flows View all (12)

NAME	TYPE	CREATED BY	LAST MODIFIED	ACTIONS
Great Outdoor Customer Orders.csv_shaped.csv_flow	Data flow	Jean-Claude Mamou	8 May 2018, 11:44:33 am	⋮
Auto Insurance Customers.csv_flow	Data flow	Carmen Ruppach	19 Apr 2018, 7:46:18 pm	⋮
Great Outdoor Customer Orders.csv_flow	Data flow	Carmen Ruppach	18 Apr 2018, 5:45:39 pm	⋮
Db2 Warehouse-WatsonDemoEnv-AUTO_INSURANCE-POLICIES_flow	Data flow	OLENA WOOLF	21 Mar 2018, 7:59:11 pm	⋮
Auto Insurance Customers Protected_flow	Data flow	Sonali Surange	19 Mar 2018, 10:03:57 pm	⋮
Auto Insurance Claims Governed Flow	Data flow	ibmid-270000muj	10 Mar 2018, 5:46:14 pm	⋮
Car Mappings_flow	Data flow	Dirk deRoos	16 Mar 2018, 3:09:26 am	⋮

Figure 8-19 Overview of project services

5. This module focuses on the VR service, so upload the created data sets to the Watson studio so they are automatically available within Watson studio on the linked object storage, as shown in Figure 8-20.



car_damage_accident_positive_examples.zip
car-damage_negatives.zip
car_damage_flat_tire_positive_examples.zip
car_damage_broken_window_positive_examples.zip
car_damage_broken_mirror_positive_examples.zip

Drop files here or browse for files to upload.

Stay on the page until upload completes. Incomplete uploads are cancelled

car_damage_accident_positive_examples.z
Pending Dismiss

car-damage_negatives.zip
Uploading file... 4 % Cancel

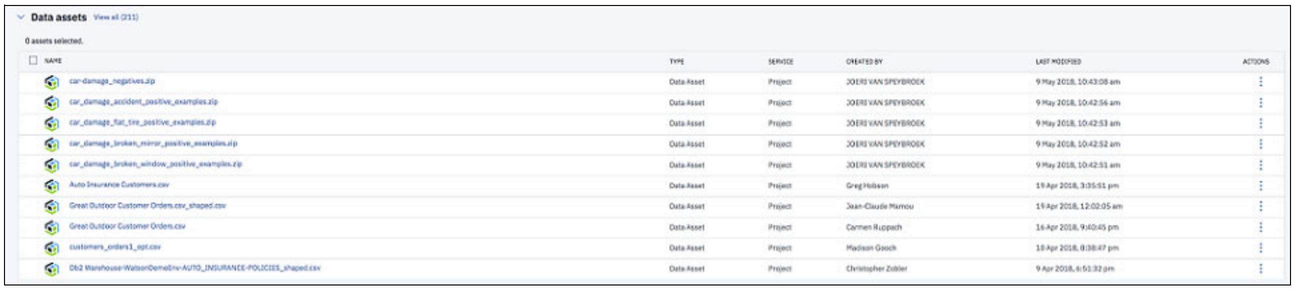
car_damage_flat_tire_positive_examples.zi
Uploading file... 8 % Cancel

car_damage_broken_window_positive_exa
Pending Dismiss

car_damage_broken_mirror_positive_exam
Pending Dismiss

Figure 8-20 Data sets available in Watson Studio

6. After the packages are uploaded, you can use them from the Watson Studio instance, as shown in Figure 8-21.

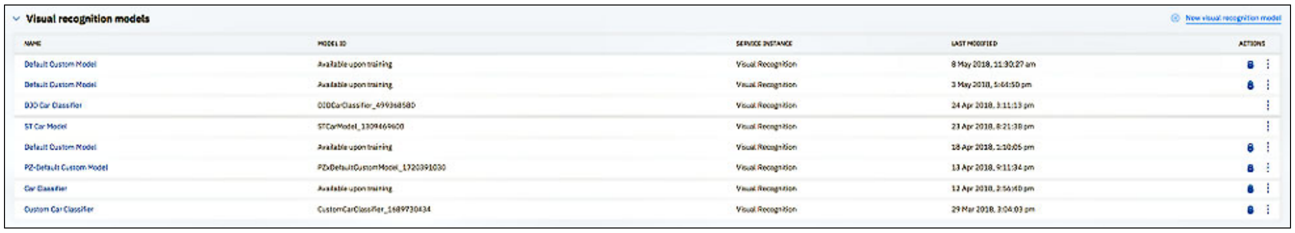

Data assets View all (211)

0 assets selected.

NAME	TYPE	SERVICE	CREATED BY	LAST MODIFIED	ACTIONS
car-damage_negatives.zip	Data Asset	Project	JOERI VAN SPEYBROEK	9 May 2018, 10:43:08 am	⋮
car_damage_accident_positive_examples.zip	Data Asset	Project	JOERI VAN SPEYBROEK	9 May 2018, 10:42:56 am	⋮
car_damage_flat_tire_positive_examples.zip	Data Asset	Project	JOERI VAN SPEYBROEK	9 May 2018, 10:42:53 am	⋮
car_damage_broken_mirror_positive_examples.zip	Data Asset	Project	JOERI VAN SPEYBROEK	9 May 2018, 10:42:52 am	⋮
car_damage_broken_window_positive_examples.zip	Data Asset	Project	JOERI VAN SPEYBROEK	9 May 2018, 10:42:51 am	⋮
Auto Insurance Customers.csv	Data Asset	Project	Greg Hobson	19 Apr 2018, 3:35:51 pm	⋮
Great Outdoor Customer Orders.csv_shaped.csv	Data Asset	Project	Jean-Claude Mamou	19 Apr 2018, 12:02:05 am	⋮
Great Outdoor Customer Orders.csv	Data Asset	Project	Carmen Ruppach	16 Apr 2018, 9:40:45 pm	⋮
customers_orders1_opt.csv	Data Asset	Project	Madison Gooch	10 Apr 2018, 8:38:47 pm	⋮
Db2 Warehouse-WatsonDemoEnv-AUTO_INSURANCE-POLICIES_shaped.csv	Data Asset	Project	Christopher Zobler	9 Apr 2018, 6:51:32 pm	⋮

Figure 8-21 Packages uploaded in Watson Studio

7. Select **New Visual Recognition Model** at the right to add a model, as shown in Figure 8-22.



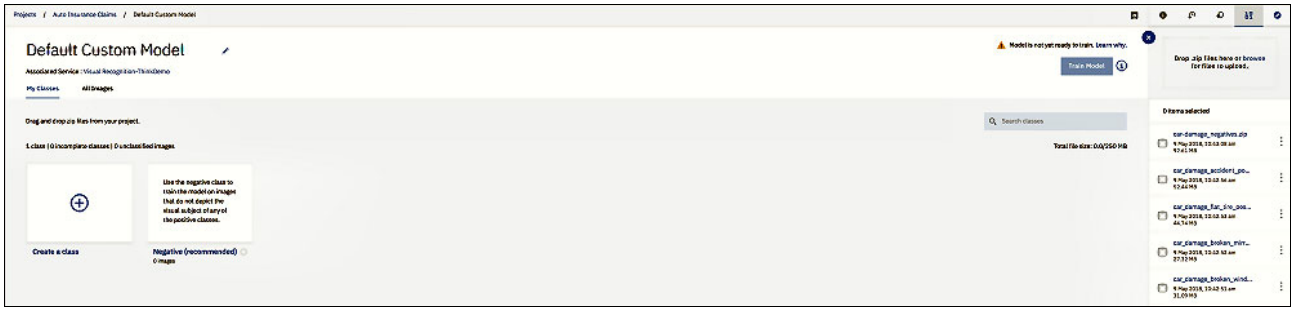
Visual recognition models

New visual recognition model

NAME	MODEL ID	SERVICE INSTANCE	LAST MODIFIED	ACTIONS
Default Custom Model	Available upon training	Visual Recognition	8 May 2018, 11:30:27 am	
Default Custom Model	Available upon training	Visual Recognition	3 May 2018, 5:64:50 pm	
DJO Car Classifier	DJOCarClassifier_499368580	Visual Recognition	24 Apr 2018, 3:11:13 pm	
ST Car Model	STCarModel_1309469600	Visual Recognition	23 Apr 2018, 8:21:38 pm	
Default Custom Model	Available upon training	Visual Recognition	18 Apr 2018, 2:10:05 pm	
PZ-Default Custom Model	PZxDefaultCustomModel_1720391030	Visual Recognition	13 Apr 2018, 9:11:34 pm	
Car Classifier	Available upon training	Visual Recognition	12 Apr 2018, 2:56:40 pm	
Custom Car Classifier	CustomCarClassifier_1689730434	Visual Recognition	29 Mar 2018, 3:04:03 pm	

Figure 8-22 Adding a model

8. Change the name and you are ready to add new classes, as shown in Figure 8-23. As when you use the API, use a single Negative Class and multiple positive classes.



Projects / Auto Insurance Claims / Default Custom Model

Default Custom Model

Associated Service : Visual Recognition-ThinkDemo

My Classes   All Images

Model is not yet ready to train. Learn why.

Train Model

Drop .zip files here or browse for files to upload.

Drag and drop zip files from your project.

1 class | 0 incomplete classes | 0 unclassified images

Search classes

Total file size: 0.0/250 MB

Create a class

Use the negative class to train the model on images that do not depict the visual subject of any of the positive classes.

Negative (recommended)
0 images

0 items selected

car-damage_negatives.zip

car_damage_accident_po...

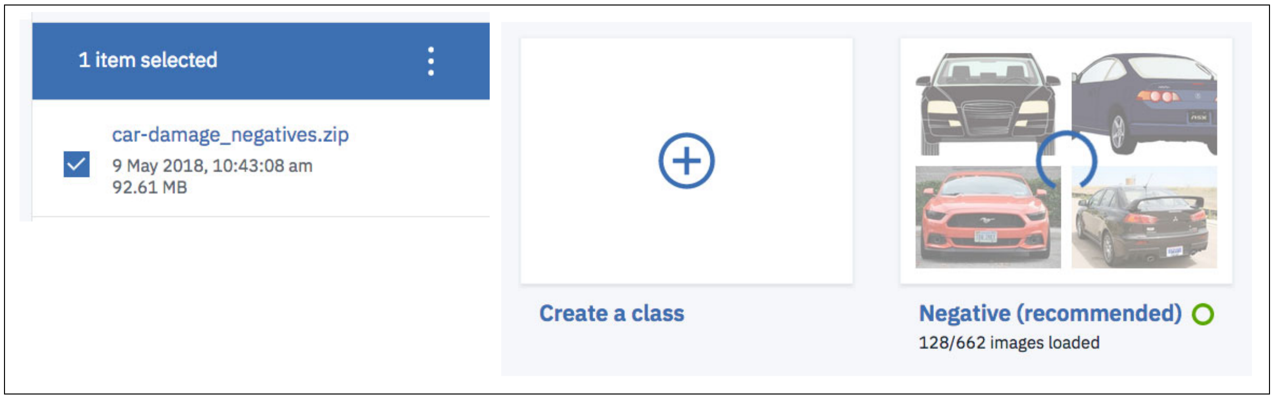
car_damage_flat_tire_pos...

car_damage_broken_mirr...

car_damage_broken_wind...

Figure 8-23 Adding classes

9. Select your negatives from the data assets on the right and drag them to the **Negative (Recommended)** class, as shown in Figure 8-24. This action starts extracting the compressed file to the Negative Class.



1 item selected

car-damage_negatives.zip
9 May 2018, 10:43:08 am
92.61 MB

Create a class

Negative (recommended)
128/662 images loaded

Figure 8-24 Selecting negatives

10. Click **Create a class** and provide the new class an appropriate name, then drag the data asset to the class so it can extract the pictures to this class, as shown in Figure 8-25.

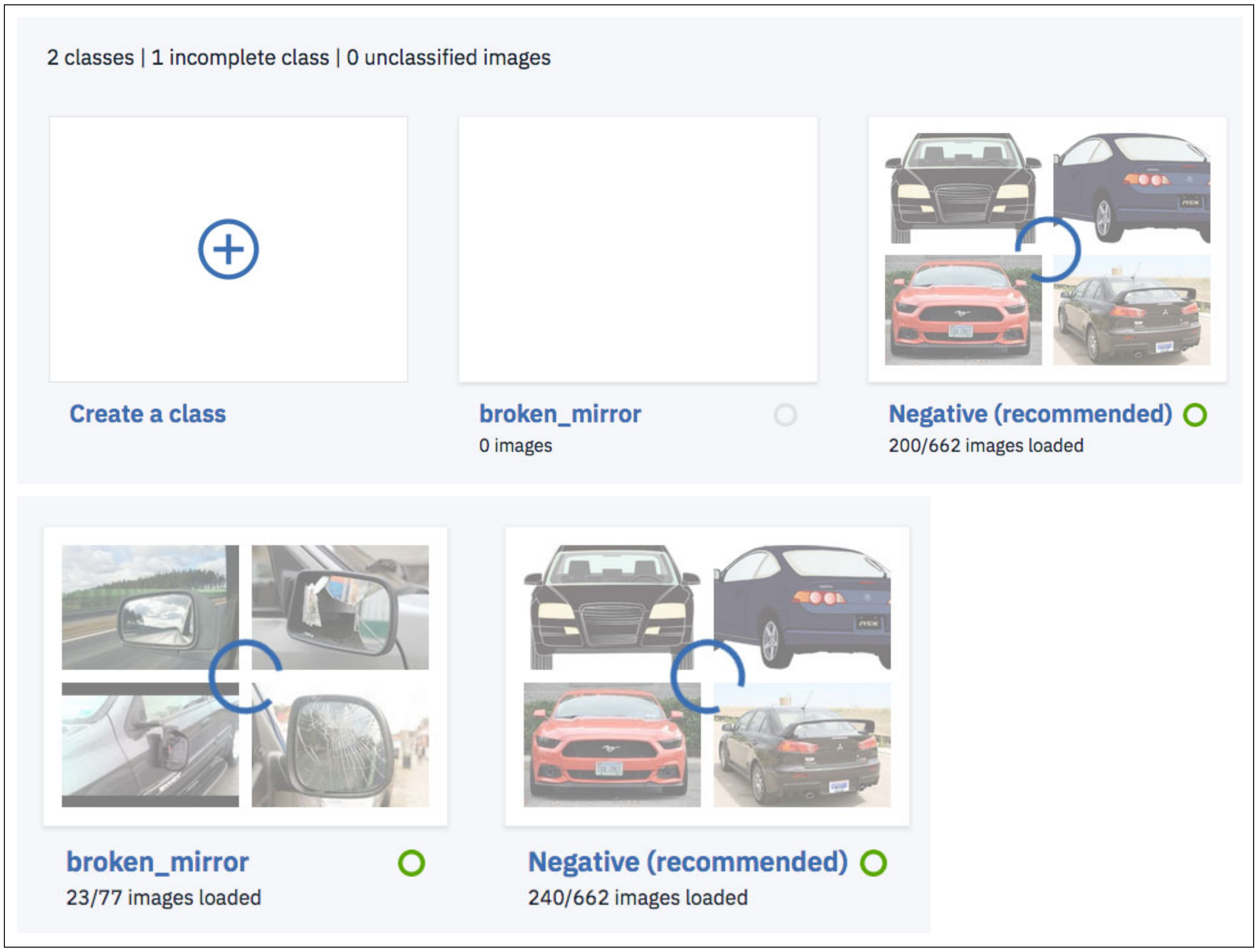


Figure 8-25 Classifying the data assets

11. After you have uploaded your images (Figure 8-26), you can start training.

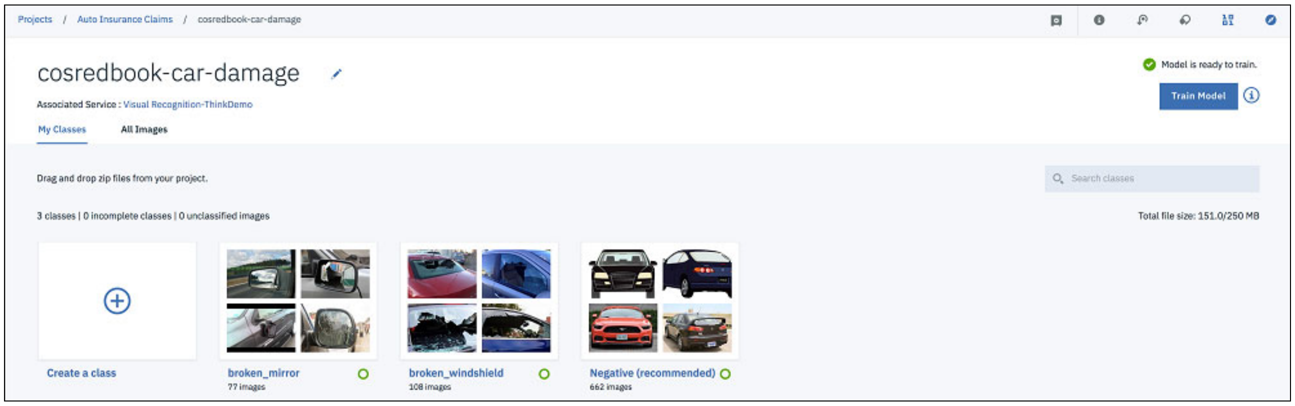
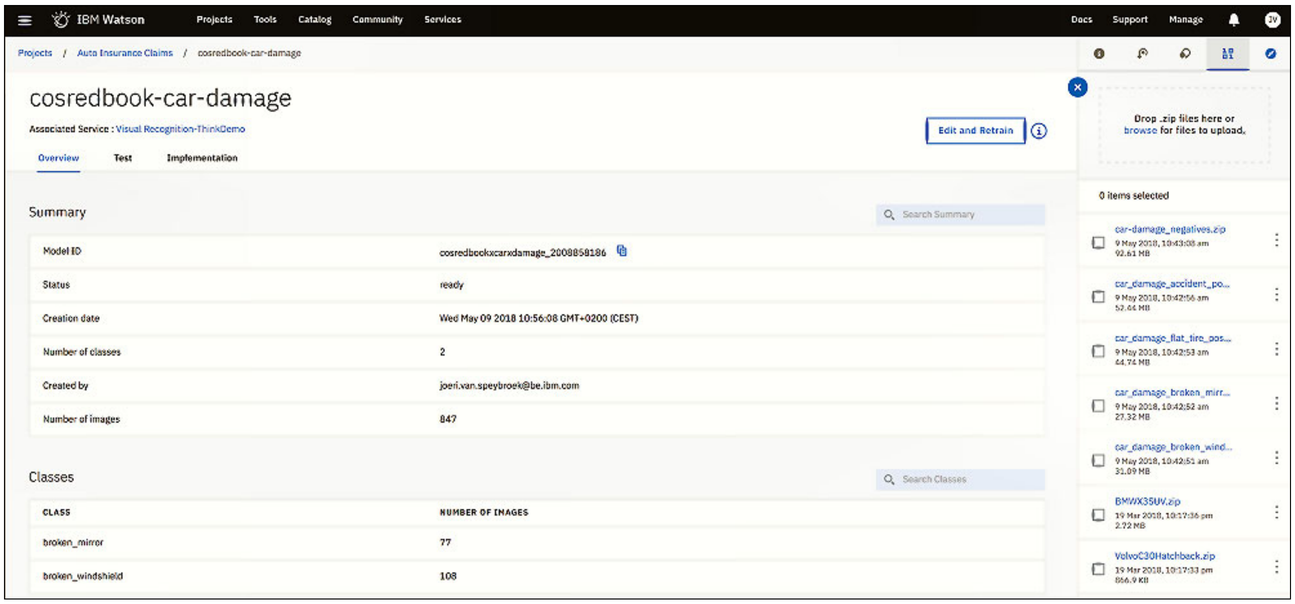

Projects / Auto Insurance Claims / cosredbook-car-damage
cosredbook-car-damage
Associated Service : Visual Recognition-ThinkDemo
My Classes
All Images
Model is ready to train.
Train Model
Drag and drop zip files from your project.
Search classes
3 classes | 0 incomplete classes | 0 unclassified images
Total file size: 151.0/250 MB
Create a class
broken_mirror
77 images
broken_windshield
108 images
Negative (recommended)
662 images

Figure 8-26 Uploading your images

12. After some time (depending on the number of images/classes), you will receive an alert that the training is complete, as shown in Figure 8-27.



cosredbook-car-damage

Associated Service : Visual Recognition-ThinkDemo

Summary

Model ID	cosredbookxcarxdamage_2008858186
Status	ready
Creation date	Wed May 09 2018 10:56:08 GMT+0200 (CEST)
Number of classes	2
Created by	joeri.van.speybroek@be.ibm.com
Number of images	847

Classes

CLASS	NUMBER OF IMAGES
broken_mirror	77
broken_windshield	108

car-damage_negatives.zip
9 May 2018, 10:43:08 am
92.61 MB

car_damage_accident_po...
9 May 2018, 10:42:56 am
52.44 MB

car_damage_flat_tire_pos...
9 May 2018, 10:42:53 am
44.74 MB

car_damage_broken_mirr...
9 May 2018, 10:42:52 am
27.32 MB

car_damage_broken_wind...
9 May 2018, 10:42:51 am
31.09 MB

BMWX35UV.zip
19 Mar 2018, 10:17:36 pm
2.72 MB

VolvoC30Hatchback.zip
19 Mar 2018, 10:17:33 pm
866.9 KB

Figure 8-27 Training completed

You can always use this module through the API using the Model ID, as shown in Example 8-22.

Example 8-22 Using the module through the API

```
curl - X GET--header 'Accept: application/json'

'https://watson-api-explorer.mybluemix.net/visual-recognition/api/v3/classifiers/
  classifier-name?
  api_key=your-api-key&
  version=2018-03-19'
```

Example 8-23 shows the expected results.

Example 8-23 Results of curl command

```
{
    "classifier_id": "cosredbookxcarxdamage_2008858186",
    "name": "cosredbook-car-damage",
    "status": "ready",
    "owner": "10ee4bc4-bc5e-4a55-8258-740998a8b5e5",
    "created": "2018-05-09T08:56:08.993Z",
    "updated": "2018-05-09T08:56:08.993Z",
    "classes": [{
        "class": "broken_windshield"
    }, {
        "class": "broken_mirror"
    }],
    "core_ml_enabled": true
}
```

However, perform the upcoming tests within Watson Studio, which offers a superior look-and-feel while also integrating this with all other services. Select **Test** so you can test your model against some pictures, as shown in Figure 8-28.

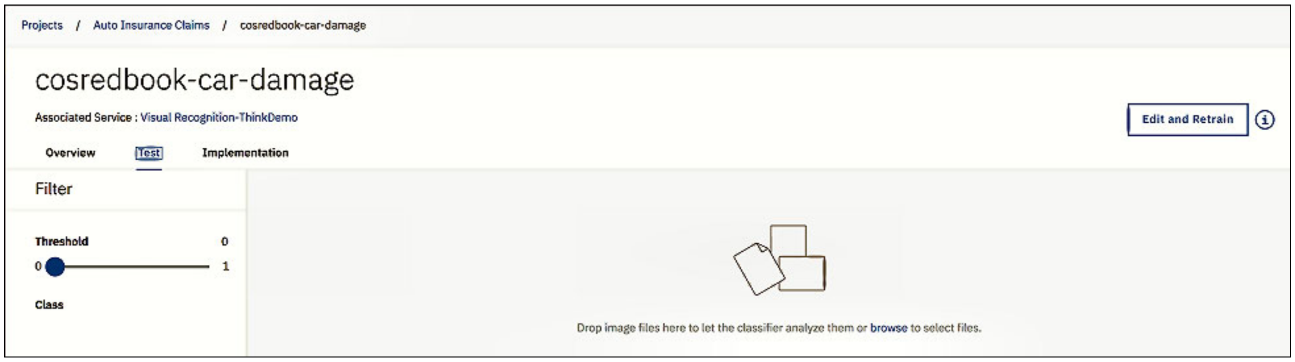

Projects / Auto Insurance Claims / cosredbook-car-damage
cosredbook-car-damage
Associated Service : Visual Recognition-ThinkDemo
Edit and Retrain
Overview
Test
Implementation
Filter
Threshold
0
0
1
Class
Drop image files here to let the classifier analyze them or browse to select files.

Figure 8-28 Performing test

Drag/Drop your test picture to the middle area to begin automatic processing of your image, as shown in Figure 8-29.

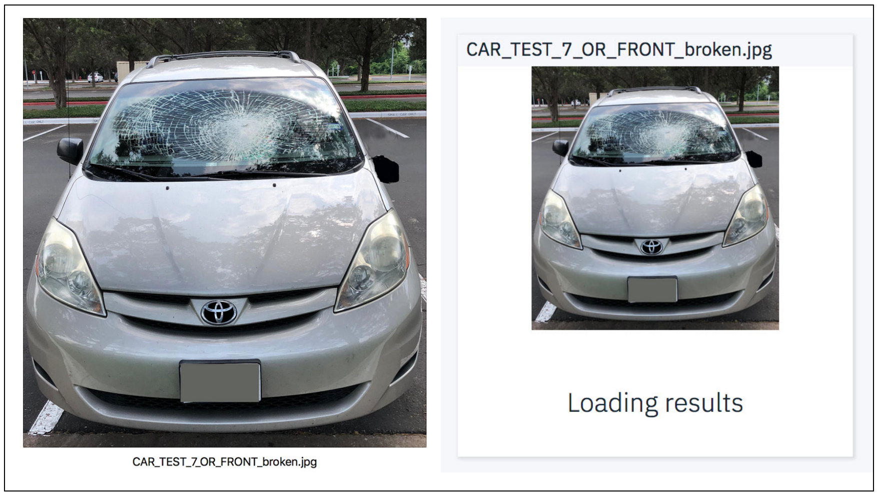

CAR_TEST_7_OR_FRONT_broken.jpg
CAR_TEST_7_OR_FRONT_broken.jpg
Loading results

Figure 8-29 Testing your picture

After a few seconds, the results are shown below the test image. In this example, one of the pictures has had a broken windshield and a broken mirror digitally added. However, the model is not confident about identifying either type of damage, as shown in Figure 8-30.

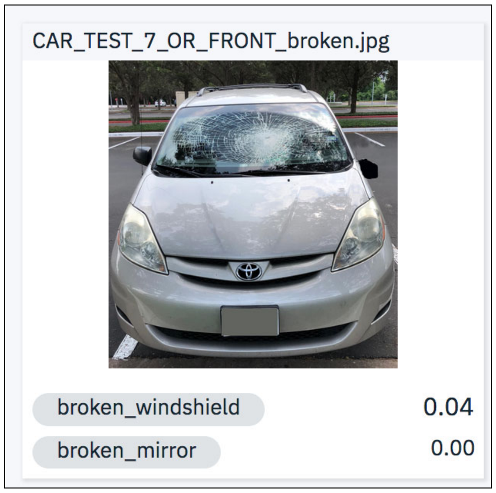


Figure 8-30 Test results

The reason for this lack of confidence is that the system has been trained on pictures that are much more close-up. The system has a hard time evaluating the same damage on a global car picture. You can improve the recognition percentage by implementing a cropping tool that creates multiple areas out of your pictures which can then be analyzed using a tagging service. The images in Figure 8-31 show manual cropping that we did that already provides much better results, although the reflection of the sun/trees in the windshield is also seen as a broken windshield, so there is still some work to do.

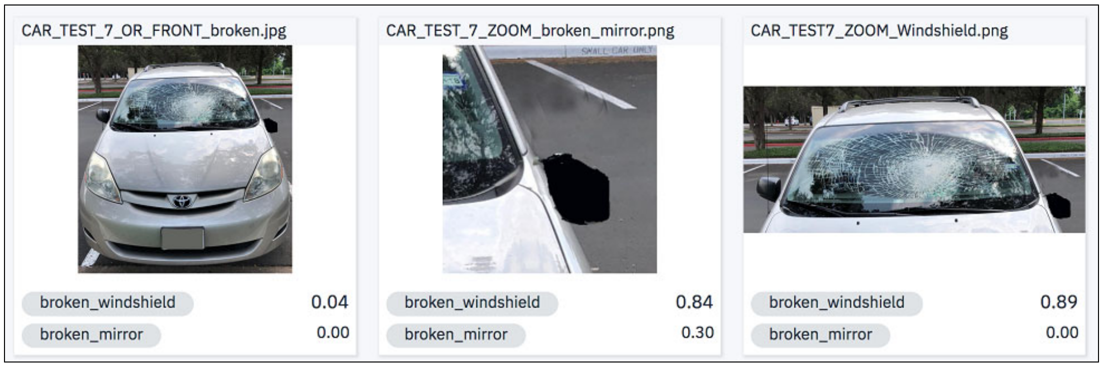


Figure 8-31 Close-up pictures also used in analysis

It is hard to create a model that can successfully analyze every zoom level of a picture in every weather condition. Therefore, when creating your own algorithms, try to remove all of these variables in the preparation phase before you test the images against your model. A good cropping tool within your application can help boost your probability scores. Implement this function in your application with the code shown in Example 8-24.

Example 8-24 Implementing a cropping tool

```
/*
The _get_car_damage is our second self-trained class with self found/created data.
This class requires does not require a paid subscription since we will only use 1
custom class.
```

```
Make sure to first follow the training steps before enabling this feature within
your code. Update the classifier ids to make it work.

This function uses our self-created damage classifier and a threshold at 0 to
allow low detection rate to still be added to the array.
We do not need to filter the results since there are only x classes (depends how
many you add), 1 per type of damage and we want to keep all withing the returned
array.
We sort the array with highest probability first where the first element of the
array [0] will be used as automatic damage detection f.
*/

function _get_car_damage(image) {
   return new Promise(function(resolve, reject) {

      var cardam_classifier_ids = ["cosredbook_car_damage_1744820941"];
      var cardam_threshold = 0;

      var params_cardam = {
         images_file: image,
         classifier_ids: cardam_classifier_ids,
         threshold: cardam_threshold
      };

      visualRecognition.classify(params_cardam, function(err, response) {
         if (err) {
            reject(err);
         }
         else {
            var car_damage = _.orderBy(response.images[0].classifiers[0].classes,
["score", "class"], ["desc", "asc"]);
            resolve(car_damage);
         }
      });
   });
}
```

8.4.8 Deploy front-end code

To deploy front-end code, complete these steps:

1. Open the code that you downloaded to `IBM-Cloud-Object-Storage-Tutorials/server/routers/claims/postClaimImage.js`.
2. Define an array that will contain all the visual recognition obtained data, as shown in Example 8-25.

Example 8-25 Creating the visual recognition data array

```
const append = {
      claimId: req.params.claimId,
      images: [imageContainer]
   };
   image.vrClassification = {};

   // create a timestamp to use when uploading the images
   let dateTime = Date.now();
```

3. Include the visual recognition actions to the nested promises. First, verify whether the image shows a car. If there is no class that even contains the word “car,” return a message asking the user to not try to fool the system, as shown in Example 8-26.

Example 8-26 Verifying that the image is of a car

```
.then(() => {
         logger.info("Uploaded original image to COS");

         // resize the image to get an normalized thumbnail
         return imageProcessor.resize(file.path);
      })
      .then(resizedImg => {
         return vrService.validateCar(resizedImg)
            .then(isCarProbability => {
               if (isCarProbability.length === 0) {
                  throw new Error("Sorry, This is not a car! Nice try though
;)");
               } else if (isCarProbability[0].score < 0.5) {
                  throw new Error("Sorry, we don't think this is a car. Nice
try though ;)");
               }

               return vrService.getCarDamage(resizedImg);
            })
```

4. Now call the car damage service to review the car for any damage that can be automatically detected, as shown in Example 8-27. Note that you will need to enhance this data set because it is not trained for uncropped images.

Example 8-27 Enhancing the data set

```
.then(carDamage => {
  logger.info("CAR DAMAGE", carDamage[0].class, carDamage[0].score);
  image.vrClassification.carDamage = carDamage;

  return vrService.getCarColor(resizedImg);
})
```

5. Next, validate the car color with the color you have in the car registration. Obtain the array of detected colors and add it to the data pool, as shown in Example 8-28.

Example 8-28 Adding the color array to the data pool

```
.then(carColor => {
  logger.info("CAR COLOR", carColor[0].class, carColor[0].score);
  image.vrClassification.carColor = carColor;

  // get the position of the car in the image using the VR service
  return vrService.getCarPosition(resizedImg);
})
```

6. The last nested promise is the car angle.

> **Note:** This promise requires a paid account. Therefore, if you are using a Lite account, only have one angle configured or leave out this part of the code.

After obtaining the car angle, return the arrays and store the images on the object stores. When this process has completed successfully, append all the information to your Cloudant document, as shown in Example 8-29.

Example 8-29 Appending the data to your Cloudant document

```
.then(carPosition => {
  logger.info("GOT CAR POSITION");

  // add the VR classification to the claim image record
  image.vrClassification.carPosition = carPosition;

  // assign the top guess of classification for the claim
  append[carPosition[0].classes[0].class] = imageId;

  // store the normalized image in a COS bucket
  let normKey =
`normalized/${req.params.claimId}/${dateTime}/${file.filename}`;
  // store key of normalized image in claimImageRecord for later retrieval
  image.normalized = normKey;

  return cos.doCreateObject(cos.normalizedBucket, file.mimetype, normKey,
resizedImg);
})
```

You have now successfully integrated the VR service into your application.

Download the provided files if you do not have your own test data. Validate your results with the results within the tutorial that are shown in Figure 8-32.

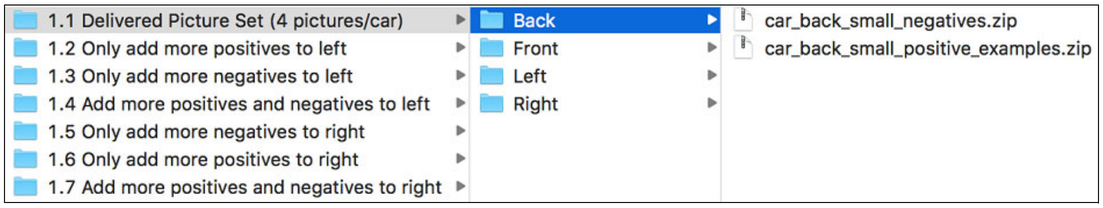

1.1 Delivered Picture Set (4 pictures/car)
1.2 Only add more positives to left
1.3 Only add more negatives to left
1.4 Add more positives and negatives to left
1.5 Only add more negatives to right
1.6 Only add more positives to right
1.7 Add more positives and negatives to right
Back
Front
Left
Right
car_back_small_negatives.zip
car_back_small_positive_examples.zip

Figure 8-32 Results of the tutorial

This module only uses 1.1 Delivered Picture Set. Module 9 covers enhancing your vr-model with sets 1.2 - 1.7.

8.5 Next steps

Now you have successfully added automation through the Watson Visual Recognition service to your application. Many other use-cases using this service can also be added, so do not hesitate to experiment further within this module.

8.6 Other references

- *Building Cognitive Applications with IBM Watson Services: Volume 1 Getting Started*, SG24-8387
- *Building Cognitive Applications with IBM Watson Services: Volume 3 Visual Recognition*, SG24-8393

9

Performing more advanced functions with machine learning

This chapter focuses on how IBM Cloud is ideal for machine learning workloads because of their peak resources requirements. An on-premises environment requires a huge investment to be able to handle the peaks. This capacity is typically 20% utilized most of the time. With the pay-per-use model, IBM Cloud provides a more economical back-end for your machine learning projects.

Machine learning is the ability to train a system to identify patterns in data based on example input. You create a model and train it with algorithms, positive and negative examples, or both so that it has enough knowledge that it can successfully recognize those patterns in any new input.

This chapter includes the following sections:

- Current architecture
- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

9.1 Introduction to machine learning

Models are extensively used in business today. For example, social media uses tagging services. Each time that you tag someone, that information is used to optimize the model so that the system can tag the same person in the future automatically.

Machine learning has improved recently because of the simplicity of gathering information for supervised learning combined with improvements in processing power that allow you to perform many tasks inline. Machine learning has existed for many decades. However, in the past it was seen as a post-process activity that took days before it produced a result.

It is important to understand that a model requires training. The better material that you provide for this training, the better results that you can expect. The first part of this module shows the negative and positive influences of training the Visual Recognition services used in the last chapter. Experiment with the Visual Recognition service to help you understand the training process of (in this case) a predefined model. There are multiple forms of training:

- Supervised Learning

  Data is labeled, and you provide negative and positive examples so that the algorithm can create its model based on them. This method is used when creating custom classes with the Visual Recognition service and is the fastest method to get high probability. However, this method also requires the most preparation.

- Unsupervised Learning

  Data is not labeled and there are no results to be interpreted. Instead, this method uses mathematical functions to define the expected results. It requires no preparation of the learning material, but is only capable in specific use-cases where results can be calculated using algorithms, such as Apriori or k-Means.

- Semi-supervised Learning

  Many real-life use-cases combine supervised and unsupervised learning. With an initial base of supervised learning, you can train your model so that when new data is obtained, the model can automatically predict the required structure to obtain the best possible outcome.

Covering all of the available algorithms is beyond the scope of this paper. However, do some research or study some open-source examples such as the notebooks below to understand how data is prepared, analyzed, and returned. For example, in our example you remove the reflection on the cars because it might be mistaken for damage by the damage detection class. The following are some common algorithms that are used in machine learning:

- Regression
  - Linear Regression
  - Logistic Regression
- Instance-based
  - k-Nearest Neighbor
  - Learning Vector Quantization
  - Locally Weighted Learning
- Regularization
  - Ridge Regression
  - Elastic Net
  - Least-Angle Regression

- Decision Tree
  - Classification and Regression Tree
  - C4.5 and C5.0
  - Conditional Decision Trees
- Bayesian
  - Naive Bayes
  - Gaussian Naive Bayes
  - Bayesian Network
- Clustering
  - k-Means
  - k-Medians
  - Hierarchical Clustering
- Association Rule
  - Apriori algorithm
  - Eclat algorithm
- Artificial Neural Network
  - Perceptron
  - Back-Propagation
  - Hopfield Network
- Deep Learning
  - Deep Belief Networks
  - Convolutional Neural Network
  - Stacked Auto-Encoders
- Dimensionality Reduction
  - Principal Component Regression (PCR)
  - Multidimensional Scaling
  - Linear Discriminant Analysis
- Ensemble Algorithm
  - Boosting
  - Stacked Generalization
  - Random Forest

Many more algorithms are available. The choice of the algorithms and the order in which you use them are crucial because the configuration affects your model performance. For example, you might want to first detect car brand and then use a brand-specific model instead of searching all car models of the world in a single classifier. Also, retrain your model from time to time with new data because the input can evolve, which requires your model to evolve. For example, you will ultimately have to add new electric cars as they are introduced so that they can be identified by your model.

IBM Cloud provides a unique machine learning experience with independent Watson modules. Watson Studio was covered in the previous chapter. This module allows you to combine different services connected to the same data sets into the same interface.

IBM Watson Machine Learning is a full-service IBM Cloud offering that makes it easy for data scientists and developers to work together to integrate predictive analytics with their applications. IBM Cloud enables organizations and developers to quickly and easily create, deploy, and manage applications on the cloud.

The focus of IBM Watson Machine Learning on IBM Cloud service is deployment. IBM SPSS Modeler or IBM Watson Studio is required to build models. IBM SPSS Modeler is a powerful, versatile data mining workbench that helps you build predictive models quickly and intuitively, without programming. Watson Studio enables you to analyze data using RStudio, Jupyter, and Python in a configured, collaborative environment that includes IBM added value, such as managed Spark and IBM Watson Machine Learning.

Both IBM SPSS Modeler and IBM Watson Studio offer various modeling methods that are not only taken from the field of machine learning, but also from the fields of artificial intelligence and statistics. All these modeling methods are included in IBM Watson Machine Learning. Therefore, IBM Watson Machine Learning Machine is not confined to modeling methods that are called "machine learning" models in the literature.

Using Cloud Object Storage with machine learning: Cloud Object Storage is an optimal storage platform for AI and machine learning workloads due to the following reasons:

- Scalability and cost efficiency: In order to deliver better algorithms, AI systems need to process vast amounts of data. Cloud Object Storage can scale limitlessly within a single namespace. It uses a dispersed storage mechanism that uses a cluster of storages nodes to store pieces of the data across the available nodes. With its low management overhead and data compression features, Cloud Object Storage provides a cost-effective way of storing large data sets required by AI algorithms.
- Cloud integration: Most of the AI systems run in the cloud. As information is created and analyzed, the storage platform should provide an easy flow of data to and from the cloud. Cloud Object Storage is optimized for cloud applications and provides an easy way to integrate Cloud and on-premise data.
- Data durability: Because AI systems use very large data sets, in many cases it is not practical or cost effective to back up this data. Instead, you need a *self-protecting* storage system. Cloud Object Storage uses an Information Dispersal Algorithm (IDA) to divide files into unrecognizable slices that are then distributed to the storage nodes. No single node has all the data, which makes it safe and less susceptible to data breaches while needing only a subset of the storage nodes to be available to fully retrieve the stored data. This ability to reassemble all the data from a subset of the chunks dramatically increases the tolerance to node and disk failures.

9.2 Current architecture

This chapter provides enhancements to the VR back-end and some notebook examples of what can be achieved with machine learning in this context. Machine learning (ML) is not implemented within the application in this module.

The following scenarios are covered in this tutorial:

- Enhance and review the re-training of our VR Service
- Create some custom notebooks to demonstrate the abilities of ML

9.3 Learning objectives

- Understand the capabilities of the Watson Machine Learning Service
- Understand the capabilities of the Watson Studio

9.4 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 8 to get the finished code from the previous module that you will build on:

```
$ cd module08
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module09
```

It would be helpful to take a Machine Learning introduction course to understand the reasoning behind the provided notebook examples.

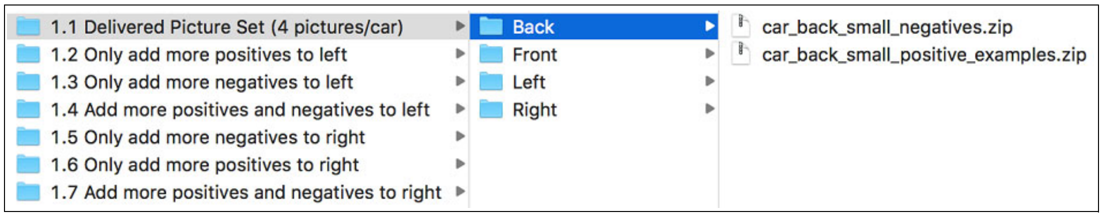
9.5 Implementation

We will present two case studies in this section:

- Enhance the VR services
- Explore more with Watson Studio: Jupyter Notebooks, Spark, and Object store

9.5.1 Enhance the VR service

Download the provided files if you do not have your own test data. Results should equal the results within the tutorial, which allows easy validation, as shown as Figure 9-1.



1.1 Delivered Picture Set (4 pictures/car)
1.2 Only add more positives to left
1.3 Only add more negatives to left
1.4 Add more positives and negatives to left
1.5 Only add more negatives to right
1.6 Only add more positives to right
1.7 Add more positives and negatives to right

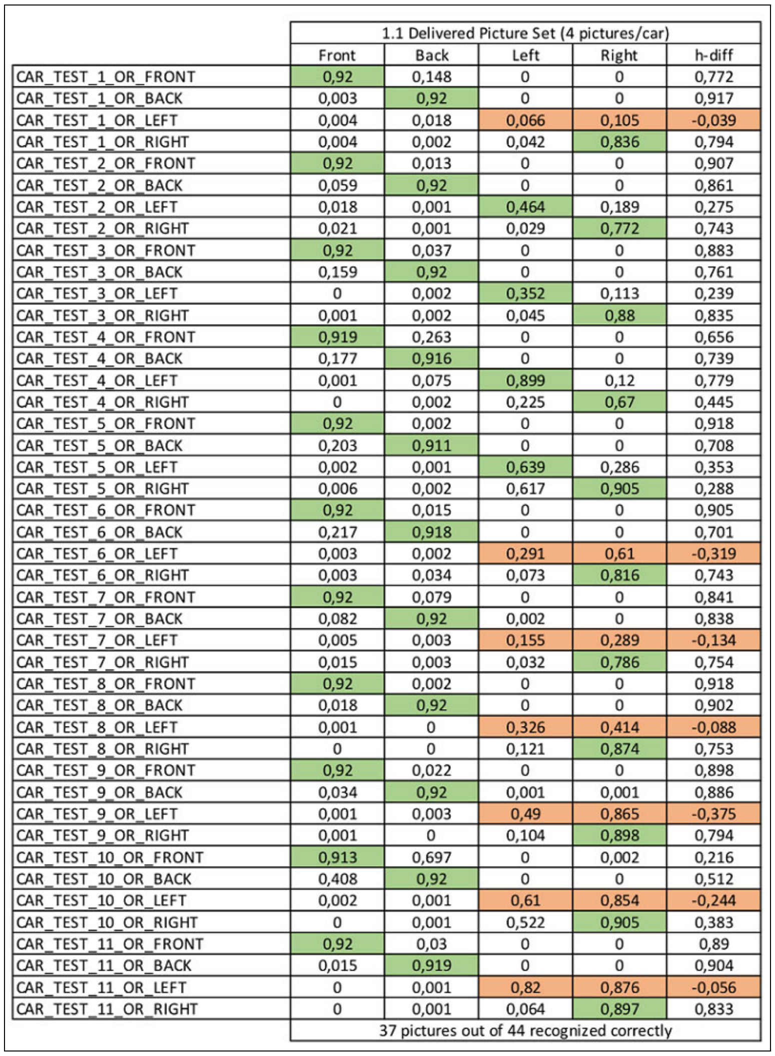
Back
Front
Left
Right

car_back_small_negatives.zip
car_back_small_positive_examples.zip

Figure 9-1 Files to be downloaded

After using data set 1.1 in the previous chapter, use the data from 1.2 - 1.7. Instead of retraining the existing custom classifier, create another with the same base so you can easily compare your results and adapt it when you are not pleased with the new results. The following examples use data sets 1.2 to 1.7, but you can create your own data sets.

Figure 9-2 shows the results generated in Module 8.

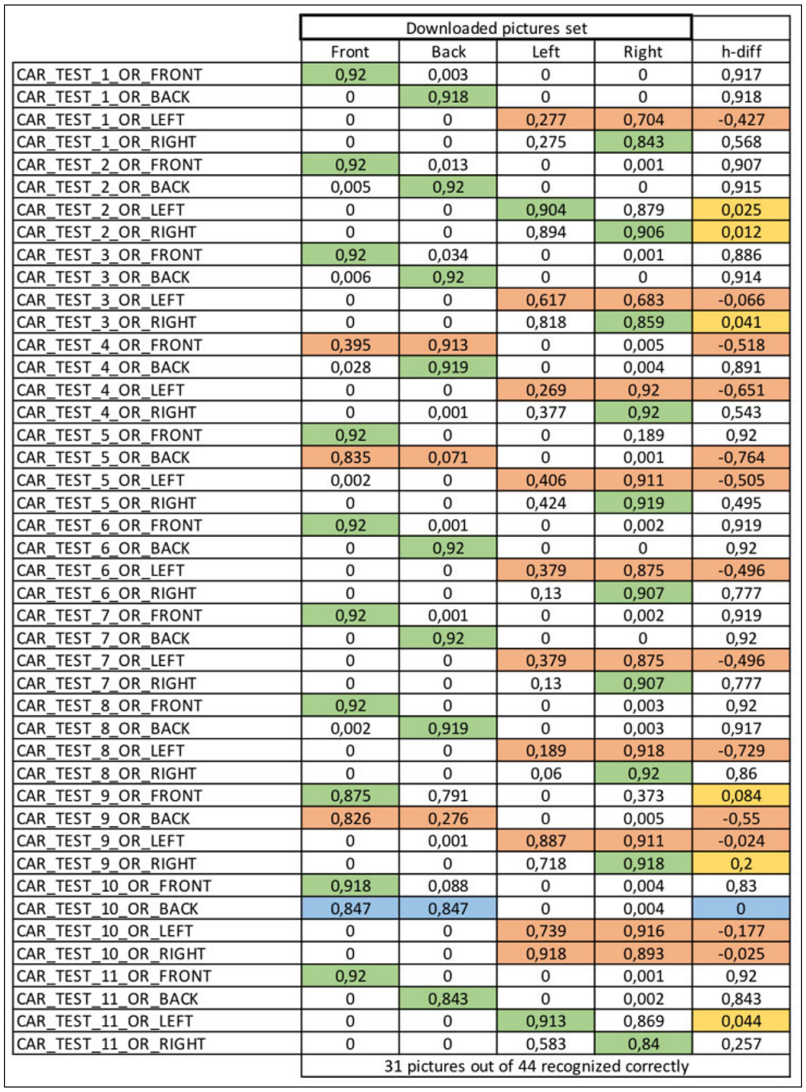


	1.1 Delivered Picture Set (4 pictures/car)				
	Front	Back	Left	Right	h-diff
CAR_TEST_1_OR_FRONT	0,92	0,148	0	0	0,772
CAR_TEST_1_OR_BACK	0,003	0,92	0	0	0,917
CAR_TEST_1_OR_LEFT	0,004	0,018	0,066	0,105	-0,039
CAR_TEST_1_OR_RIGHT	0,004	0,002	0,042	0,836	0,794
CAR_TEST_2_OR_FRONT	0,92	0,013	0	0	0,907
CAR_TEST_2_OR_BACK	0,059	0,92	0	0	0,861
CAR_TEST_2_OR_LEFT	0,018	0,001	0,464	0,189	0,275
CAR_TEST_2_OR_RIGHT	0,021	0,001	0,029	0,772	0,743
CAR_TEST_3_OR_FRONT	0,92	0,037	0	0	0,883
CAR_TEST_3_OR_BACK	0,159	0,92	0	0	0,761
CAR_TEST_3_OR_LEFT	0	0,002	0,352	0,113	0,239
CAR_TEST_3_OR_RIGHT	0,001	0,002	0,045	0,88	0,835
CAR_TEST_4_OR_FRONT	0,919	0,263	0	0	0,656
CAR_TEST_4_OR_BACK	0,177	0,916	0	0	0,739
CAR_TEST_4_OR_LEFT	0,001	0,075	0,899	0,12	0,779
CAR_TEST_4_OR_RIGHT	0	0,002	0,225	0,67	0,445
CAR_TEST_5_OR_FRONT	0,92	0,002	0	0	0,918
CAR_TEST_5_OR_BACK	0,203	0,911	0	0	0,708
CAR_TEST_5_OR_LEFT	0,002	0,001	0,639	0,286	0,353
CAR_TEST_5_OR_RIGHT	0,006	0,002	0,617	0,905	0,288
CAR_TEST_6_OR_FRONT	0,92	0,015	0	0	0,905
CAR_TEST_6_OR_BACK	0,217	0,918	0	0	0,701
CAR_TEST_6_OR_LEFT	0,003	0,002	0,291	0,61	-0,319
CAR_TEST_6_OR_RIGHT	0,003	0,034	0,073	0,816	0,743
CAR_TEST_7_OR_FRONT	0,92	0,079	0	0	0,841
CAR_TEST_7_OR_BACK	0,082	0,92	0,002	0	0,838
CAR_TEST_7_OR_LEFT	0,005	0,003	0,155	0,289	-0,134
CAR_TEST_7_OR_RIGHT	0,015	0,003	0,032	0,786	0,754
CAR_TEST_8_OR_FRONT	0,92	0,002	0	0	0,918
CAR_TEST_8_OR_BACK	0,018	0,92	0	0	0,902
CAR_TEST_8_OR_LEFT	0,001	0	0,326	0,414	-0,088
CAR_TEST_8_OR_RIGHT	0	0	0,121	0,874	0,753
CAR_TEST_9_OR_FRONT	0,92	0,022	0	0	0,898
CAR_TEST_9_OR_BACK	0,034	0,92	0,001	0,001	0,886
CAR_TEST_9_OR_LEFT	0,001	0,003	0,49	0,865	-0,375
CAR_TEST_9_OR_RIGHT	0,001	0	0,104	0,898	0,794
CAR_TEST_10_OR_FRONT	0,913	0,697	0	0,002	0,216
CAR_TEST_10_OR_BACK	0,408	0,92	0	0	0,512
CAR_TEST_10_OR_LEFT	0,002	0,001	0,61	0,854	-0,244
CAR_TEST_10_OR_RIGHT	0	0,001	0,522	0,905	0,383
CAR_TEST_11_OR_FRONT	0,92	0,03	0	0	0,89
CAR_TEST_11_OR_BACK	0,015	0,919	0	0	0,904
CAR_TEST_11_OR_LEFT	0	0,001	0,82	0,876	-0,056
CAR_TEST_11_OR_RIGHT	0	0,001	0,064	0,897	0,833
	37 pictures out of 44 recognized correctly				

Figure 9-2 Results from Module 8

This data shows that of 44 pictures of the cars, 37 sides were recognized correctly. With new pictures, there is a high probability of identifying the front and back, whereas left and right is always a struggle. You can add more angles to the existing 1.1 data set to improve the results.

In the following example, we downloaded hundreds of random pictures with a mass downloader to create a customer classifier. These images were only sorted as front/back/left/right and used to train the model. Figure 9-3 shows the results from the random set without any preparation.

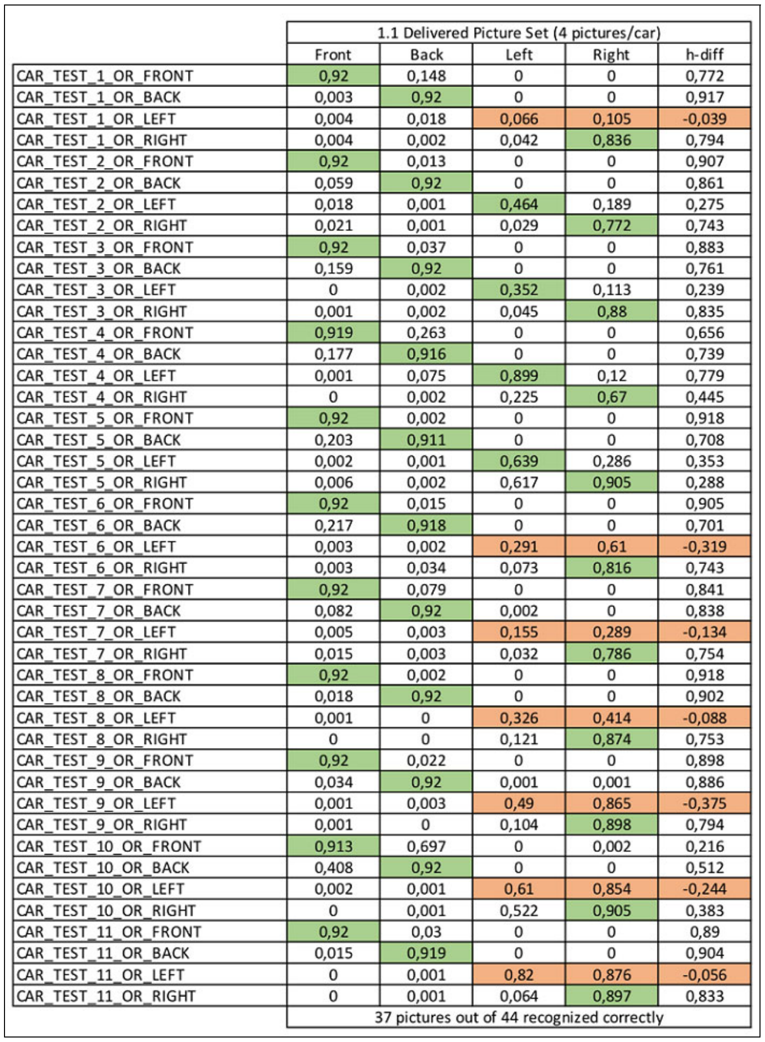


	Downloaded pictures set				
	Front	Back	Left	Right	h-diff
CAR_TEST_1_OR_FRONT	0,92	0,003	0	0	0,917
CAR_TEST_1_OR_BACK	0	0,918	0	0	0,918
CAR_TEST_1_OR_LEFT	0	0	0,277	0,704	-0,427
CAR_TEST_1_OR_RIGHT	0	0	0,275	0,843	0,568
CAR_TEST_2_OR_FRONT	0,92	0,013	0	0,001	0,907
CAR_TEST_2_OR_BACK	0,005	0,92	0	0	0,915
CAR_TEST_2_OR_LEFT	0	0	0,904	0,879	0,025
CAR_TEST_2_OR_RIGHT	0	0	0,894	0,906	0,012
CAR_TEST_3_OR_FRONT	0,92	0,034	0	0,001	0,886
CAR_TEST_3_OR_BACK	0,006	0,92	0	0	0,914
CAR_TEST_3_OR_LEFT	0	0	0,617	0,683	-0,066
CAR_TEST_3_OR_RIGHT	0	0	0,818	0,859	0,041
CAR_TEST_4_OR_FRONT	0,395	0,913	0	0,005	-0,518
CAR_TEST_4_OR_BACK	0,028	0,919	0	0,004	0,891
CAR_TEST_4_OR_LEFT	0	0	0,269	0,92	-0,651
CAR_TEST_4_OR_RIGHT	0	0,001	0,377	0,92	0,543
CAR_TEST_5_OR_FRONT	0,92	0	0	0,189	0,92
CAR_TEST_5_OR_BACK	0,835	0,071	0	0,001	-0,764
CAR_TEST_5_OR_LEFT	0,002	0	0,406	0,911	-0,505
CAR_TEST_5_OR_RIGHT	0	0	0,424	0,919	0,495
CAR_TEST_6_OR_FRONT	0,92	0,001	0	0,002	0,919
CAR_TEST_6_OR_BACK	0	0,92	0	0	0,92
CAR_TEST_6_OR_LEFT	0	0	0,379	0,875	-0,496
CAR_TEST_6_OR_RIGHT	0	0	0,13	0,907	0,777
CAR_TEST_7_OR_FRONT	0,92	0,001	0	0,002	0,919
CAR_TEST_7_OR_BACK	0	0,92	0	0	0,92
CAR_TEST_7_OR_LEFT	0	0	0,379	0,875	-0,496
CAR_TEST_7_OR_RIGHT	0	0	0,13	0,907	0,777
CAR_TEST_8_OR_FRONT	0,92	0	0	0,003	0,92
CAR_TEST_8_OR_BACK	0,002	0,919	0	0,003	0,917
CAR_TEST_8_OR_LEFT	0	0	0,189	0,918	-0,729
CAR_TEST_8_OR_RIGHT	0	0	0,06	0,92	0,86
CAR_TEST_9_OR_FRONT	0,875	0,791	0	0,373	0,084
CAR_TEST_9_OR_BACK	0,826	0,276	0	0,005	-0,55
CAR_TEST_9_OR_LEFT	0	0,001	0,887	0,911	-0,024
CAR_TEST_9_OR_RIGHT	0	0	0,718	0,918	0,2
CAR_TEST_10_OR_FRONT	0,918	0,088	0	0,004	0,83
CAR_TEST_10_OR_BACK	0,847	0,847	0	0,004	0
CAR_TEST_10_OR_LEFT	0	0	0,739	0,916	-0,177
CAR_TEST_10_OR_RIGHT	0	0	0,918	0,893	-0,025
CAR_TEST_11_OR_FRONT	0,92	0	0	0,001	0,92
CAR_TEST_11_OR_BACK	0	0,843	0	0,002	0,843
CAR_TEST_11_OR_LEFT	0	0	0,913	0,869	0,044
CAR_TEST_11_OR_RIGHT	0	0	0,583	0,84	0,257
	31 pictures out of 44 recognized correctly				

Figure 9-3 Results for the random set without preparation

Even though it is “only” 31 out of 44 pictures, this is a great result because these pictures are much different from the test pictures. Until recently, 70% was seen as a very high probability. With some preparation and modification, it should be easy to get it close to 90%.

For the next exercise, enhance your self-created model by using the car pictures. As you might have noticed, the system is struggling with high probability on right and very low on left, so there are some influences that significantly affect identifying the left side, as shown in Figure 9-4.

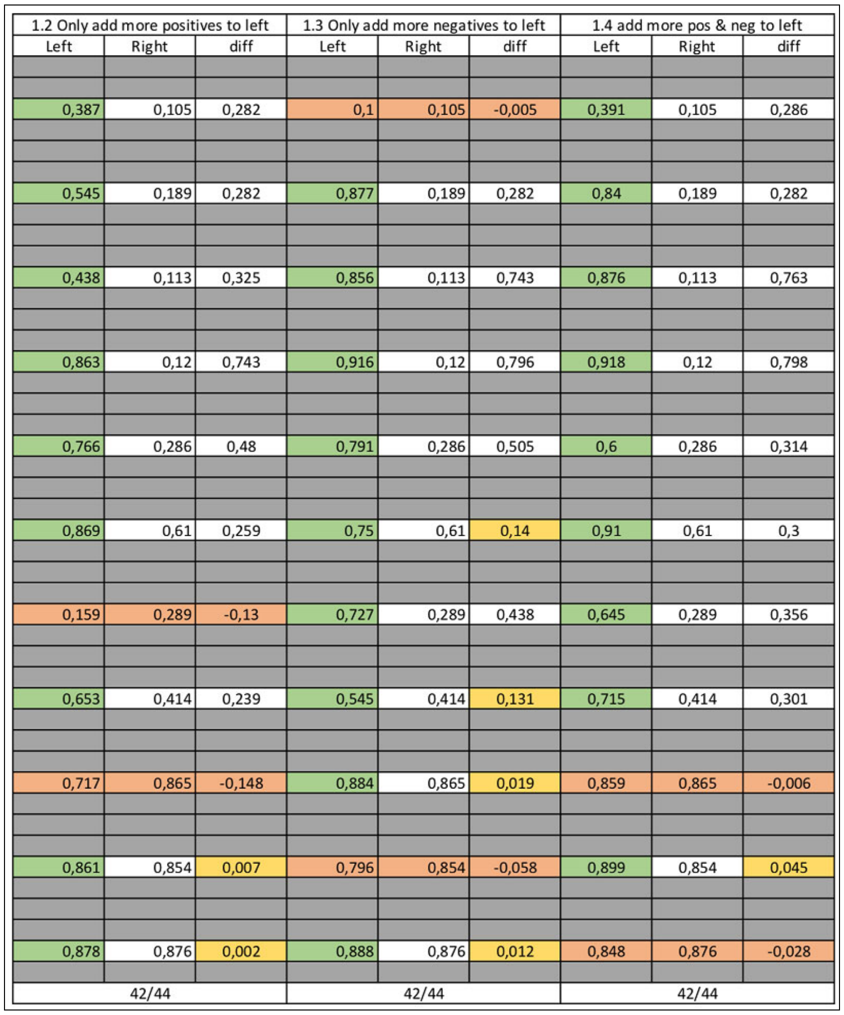


	1.1 Delivered Picture Set (4 pictures/car)				
	Front	Back	Left	Right	h-diff
CAR_TEST_1_OR_FRONT	0,92	0,148	0	0	0,772
CAR_TEST_1_OR_BACK	0,003	0,92	0	0	0,917
CAR_TEST_1_OR_LEFT	0,004	0,018	0,066	0,105	-0,039
CAR_TEST_1_OR_RIGHT	0,004	0,002	0,042	0,836	0,794
CAR_TEST_2_OR_FRONT	0,92	0,013	0	0	0,907
CAR_TEST_2_OR_BACK	0,059	0,92	0	0	0,861
CAR_TEST_2_OR_LEFT	0,018	0,001	0,464	0,189	0,275
CAR_TEST_2_OR_RIGHT	0,021	0,001	0,029	0,772	0,743
CAR_TEST_3_OR_FRONT	0,92	0,037	0	0	0,883
CAR_TEST_3_OR_BACK	0,159	0,92	0	0	0,761
CAR_TEST_3_OR_LEFT	0	0,002	0,352	0,113	0,239
CAR_TEST_3_OR_RIGHT	0,001	0,002	0,045	0,88	0,835
CAR_TEST_4_OR_FRONT	0,919	0,263	0	0	0,656
CAR_TEST_4_OR_BACK	0,177	0,916	0	0	0,739
CAR_TEST_4_OR_LEFT	0,001	0,075	0,899	0,12	0,779
CAR_TEST_4_OR_RIGHT	0	0,002	0,225	0,67	0,445
CAR_TEST_5_OR_FRONT	0,92	0,002	0	0	0,918
CAR_TEST_5_OR_BACK	0,203	0,911	0	0	0,708
CAR_TEST_5_OR_LEFT	0,002	0,001	0,639	0,286	0,353
CAR_TEST_5_OR_RIGHT	0,006	0,002	0,617	0,905	0,288
CAR_TEST_6_OR_FRONT	0,92	0,015	0	0	0,905
CAR_TEST_6_OR_BACK	0,217	0,918	0	0	0,701
CAR_TEST_6_OR_LEFT	0,003	0,002	0,291	0,61	-0,319
CAR_TEST_6_OR_RIGHT	0,003	0,034	0,073	0,816	0,743
CAR_TEST_7_OR_FRONT	0,92	0,079	0	0	0,841
CAR_TEST_7_OR_BACK	0,082	0,92	0,002	0	0,838
CAR_TEST_7_OR_LEFT	0,005	0,003	0,155	0,289	-0,134
CAR_TEST_7_OR_RIGHT	0,015	0,003	0,032	0,786	0,754
CAR_TEST_8_OR_FRONT	0,92	0,002	0	0	0,918
CAR_TEST_8_OR_BACK	0,018	0,92	0	0	0,902
CAR_TEST_8_OR_LEFT	0,001	0	0,326	0,414	-0,088
CAR_TEST_8_OR_RIGHT	0	0	0,121	0,874	0,753
CAR_TEST_9_OR_FRONT	0,92	0,022	0	0	0,898
CAR_TEST_9_OR_BACK	0,034	0,92	0,001	0,001	0,886
CAR_TEST_9_OR_LEFT	0,001	0,003	0,49	0,865	-0,375
CAR_TEST_9_OR_RIGHT	0,001	0	0,104	0,898	0,794
CAR_TEST_10_OR_FRONT	0,913	0,697	0	0,002	0,216
CAR_TEST_10_OR_BACK	0,408	0,92	0	0	0,512
CAR_TEST_10_OR_LEFT	0,002	0,001	0,61	0,854	-0,244
CAR_TEST_10_OR_RIGHT	0	0,001	0,522	0,905	0,383
CAR_TEST_11_OR_FRONT	0,92	0,03	0	0	0,89
CAR_TEST_11_OR_BACK	0,015	0,919	0	0	0,904
CAR_TEST_11_OR_LEFT	0	0,001	0,82	0,876	-0,056
CAR_TEST_11_OR_RIGHT	0	0,001	0,064	0,897	0,833
	37 pictures out of 44 recognized correctly				

Figure 9-4 Problem results while identifying the left side

Add more pictures in the same scene to your custom classifier for review to enhance your right and left classification. First, create a model using the existing 1.1 image set so you keep every classifier for easy comparison.

For the Left side, see what results you can get by adding only positive, only negative, or extra positive and negative pictures, as shown in Figure 9-5.

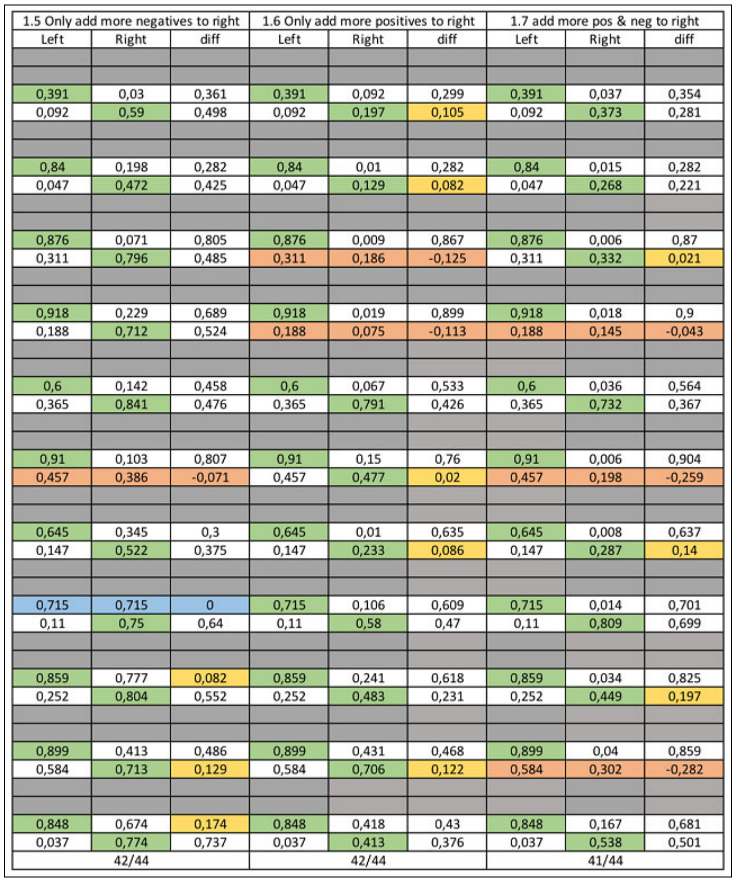


1.2 Only add more positives to left			1.3 Only add more negatives to left			1.4 add more pos & neg to left		
Left	Right	diff	Left	Right	diff	Left	Right	diff
0,387	0,105	0,282	0,1	0,105	-0,005	0,391	0,105	0,286
0,545	0,189	0,282	0,877	0,189	0,282	0,84	0,189	0,282
0,438	0,113	0,325	0,856	0,113	0,743	0,876	0,113	0,763
0,863	0,12	0,743	0,916	0,12	0,796	0,918	0,12	0,798
0,766	0,286	0,48	0,791	0,286	0,505	0,6	0,286	0,314
0,869	0,61	0,259	0,75	0,61	0,14	0,91	0,61	0,3
0,159	0,289	-0,13	0,727	0,289	0,438	0,645	0,289	0,356
0,653	0,414	0,239	0,545	0,414	0,131	0,715	0,414	0,301
0,717	0,865	-0,148	0,884	0,865	0,019	0,859	0,865	-0,006
0,861	0,854	0,007	0,796	0,854	-0,058	0,899	0,854	0,045
0,878	0,876	0,002	0,888	0,876	0,012	0,848	0,876	-0,028
	42/44			42/44			42/44	

Figure 9-5 Enhancing the Left side results

The results show that you are not only comparing the pictures, but creating an algorithm. We get the same 42 out of the 44 pictures correctly classified, but different pictures are seen as false positives/negatives. Therefore, by adding random pictures from different cars, you are enhancing your algorithm and obtaining better results. You might notice that there is still room for improvement because your differentials are low in multiple images.

Determine if adding more images to the right side allows you to obtain even better results (Figure 9-6).



1.5 Only add more negatives to right			1.6 Only add more positives to right			1.7 add more pos & neg to right		
Left	Right	diff	Left	Right	diff	Left	Right	diff
0,391	0,03	0,361	0,391	0,092	0,299	0,391	0,037	0,354
0,092	0,59	0,498	0,092	0,197	0,105	0,092	0,373	0,281
0,84	0,198	0,282	0,84	0,01	0,282	0,84	0,015	0,282
0,047	0,472	0,425	0,047	0,129	0,082	0,047	0,268	0,221
0,876	0,071	0,805	0,876	0,009	0,867	0,876	0,006	0,87
0,311	0,796	0,485	0,311	0,186	-0,125	0,311	0,332	0,021
0,918	0,229	0,689	0,918	0,019	0,899	0,918	0,018	0,9
0,188	0,712	0,524	0,188	0,075	-0,113	0,188	0,145	-0,043
0,6	0,142	0,458	0,6	0,067	0,533	0,6	0,036	0,564
0,365	0,841	0,476	0,365	0,791	0,426	0,365	0,732	0,367
0,91	0,103	0,807	0,91	0,15	0,76	0,91	0,006	0,904
0,457	0,386	-0,071	0,457	0,477	0,02	0,457	0,198	-0,259
0,645	0,345	0,3	0,645	0,01	0,635	0,645	0,008	0,637
0,147	0,522	0,375	0,147	0,233	0,086	0,147	0,287	0,14
0,715	0,715	0	0,715	0,106	0,609	0,715	0,014	0,701
0,11	0,75	0,64	0,11	0,58	0,47	0,11	0,809	0,699
0,859	0,777	0,082	0,859	0,241	0,618	0,859	0,034	0,825
0,252	0,804	0,552	0,252	0,483	0,231	0,252	0,449	0,197
0,899	0,413	0,486	0,899	0,431	0,468	0,899	0,04	0,859
0,584	0,713	0,129	0,584	0,706	0,122	0,584	0,302	-0,282
0,848	0,674	0,174	0,848	0,418	0,43	0,848	0,167	0,681
0,037	0,774	0,737	0,037	0,413	0,376	0,037	0,538	0,501
	42/44			42/44			41/44	

Figure 9-6 Enhancing the right side results

Apparently we get the best results by only adding extra negatives to the right side. Therefore, add examples that could be used as positives to the left as negatives to the right instead. Even though we still “only” get 42.5 out of 44, this still represents the best results we have achieved. Even just adding positive and negative images to the same algorithm reduces overall performance, which shows that proper data preparation is important.

One of the problems in this picture set is that some pictures do not have the correct 90-degree angle. Instead of enhancing your algorithm, they have the opposite effect of adding doubt. We saw a similar issue with damage detection. In this case, it is important to remove external flare and reflections from the images because these might be mistaken for damage by the algorithm.

The limitation of using a pre-built VR service is that you cannot introduce extra data preparation. You can combine Watson Studio with other machine learning tools. This is not in scope for this module, but you can search for some examples and combine image processing capabilities (such as remove color, darken/lighten, and so on) with the VR service to obtain higher probability.

9.5.2 Explore more with Watson Studio, including Jupyter Notebooks, Spark, and Object store

In Module 8, you had a brief interaction with Watson Studio. Watson Studio is an advanced social engineering platform that brings together data scientists, data engineers, and machine learning experts to work collaboratively on projects using the best tools and technologies in an intuitive manner. You have seen how the Watson Visual Recognition Service is used in Watson Studio.

This section focuses on other kinds of analytics that can be performed using Juptyer notebooks and Apache Spark libraries with data on an object store.

Auto Insurance Claims Analytics using Analytics Engine

In accordance with our example use case, this exercise demonstrates how you can create a simple chart of the number of auto insurance claims filed in Insurance Company X. It gives the total count of insurance claims against the reason for claim such as hail, accident, and so on. It provides a step-by-step process to create a final bar chart by creating a new Jupyter Notebook in Watson Studio.

Create a project in WatsonStudio and associate services

After you have logged in to the Watson Studio dashboard, you can create a project. As with any project creation in the data science platform UI, you can associate a service with the project in the Settings tab. We have chosen to associate the Analytics Engine service from the IBM Cloud catalog. Analytics Engine is a powerful IBM Cloud service that allows you to run Spark and Hadoop workloads against your data. This is the engine that will perform the Spark analytics for this example. You can create an instance as you create a project or you can work with a previously created instance. This link gives a good overview of how you can do this.

Associate an instance of Object Store service with the project

You can also associate an object store with the project. Again, the UI automatically discovers instances of object store if you have already created them. Otherwise, you can create one. When you associate an object store, all of the assets that are associated with the project automatically get stored in the object store, transparently. Some of the code templates that are generated are also based on this object store, as you will see below.

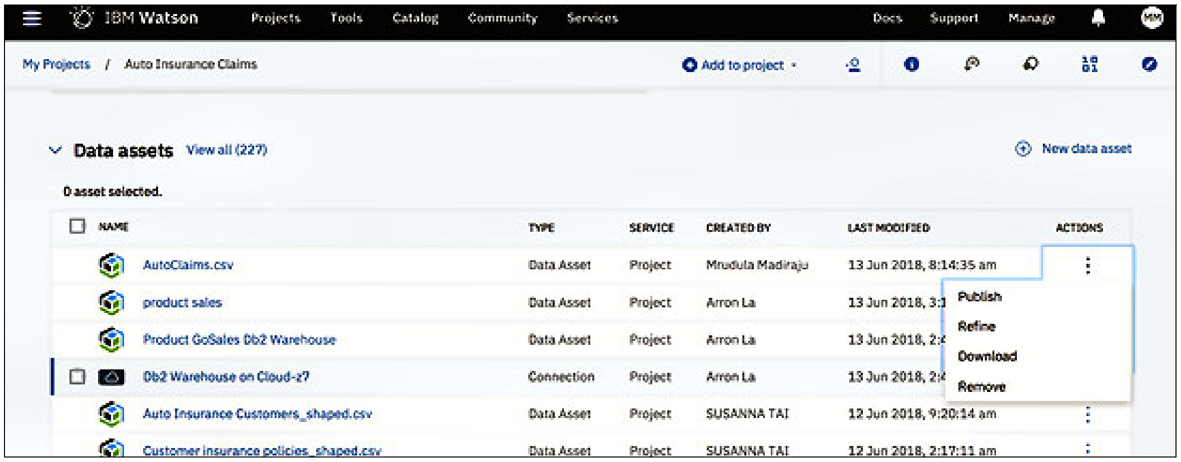
Upload a data set as an "asset"

This tutorial involves publicly available Auto Insurance Claims - Automobile Insurance claims including location, policy type, and claim amount. Download this file as a `.csv` and upload it into the Assets section of the project so that it is available for use within the notebook.

Refine the data set

Use the Data Refinery service to change this raw data. Data Refinery service is typically used to cleanse, transform, and prepare your data set using simple intuitive visual tools. To learn more, see the Data Refinery website. For example, the data from the asset uploaded earlier is all considered String by default. You want to specify the data types as float, Boolean, and so on, for performing specific analytics later. Another refinement could be renaming of columns. Because there are spaces in the column names in the original sheet, you can edit and rename them as needed.

Figure 9-7 shows this process `Claim_Reason`.


IBM Watson Projects Tools Catalog Community Services Docs Support Manage

My Projects / Auto Insurance Claims Add to project

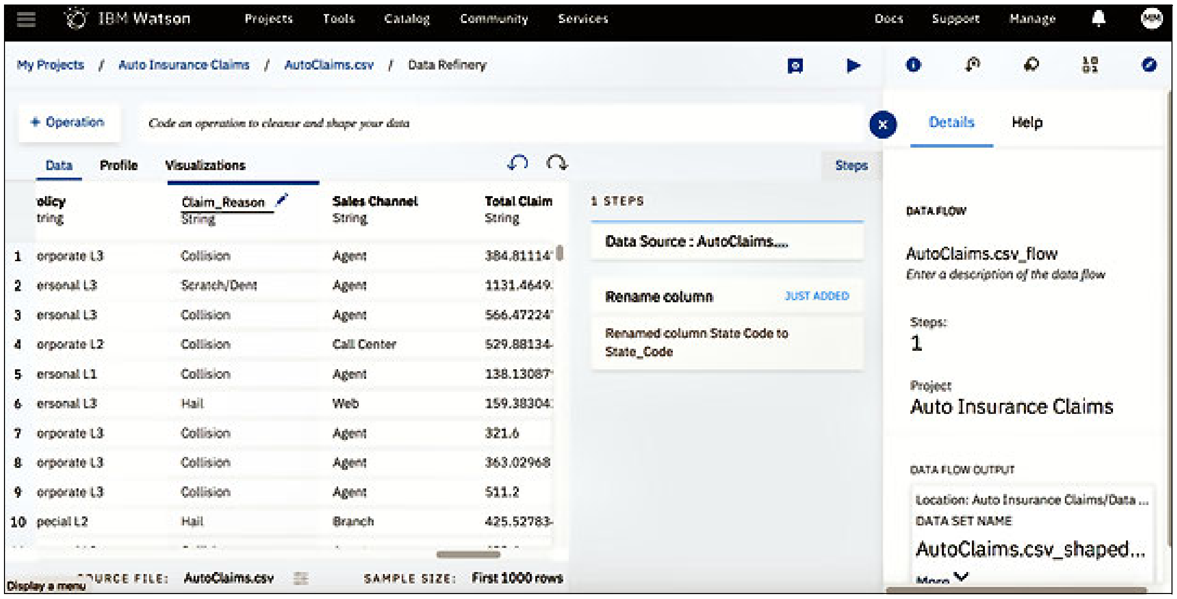
Data assets View all (227) New data asset

0 asset selected.

NAME	TYPE	SERVICE	CREATED BY	LAST MODIFIED	ACTIONS
AutoClaims.csv	Data Asset	Project	Mrudula Madiraju	13 Jun 2018, 8:14:35 am	⋮
product sales	Data Asset	Project	Arron La	13 Jun 2018, 3:1	Publish
Product GoSales Db2 Warehouse	Data Asset	Project	Arron La	13 Jun 2018, 2:4	Refine
Db2 Warehouse on Cloud-z7	Connection	Project	Arron La	13 Jun 2018, 2:4	Download / Remove
Auto Insurance Customers_shaped.csv	Data Asset	Project	SUSANNA TAI	12 Jun 2018, 9:20:14 am	⋮
Customer insurance policies_shaped.csv	Data Asset	Project	SUSANNA TAI	12 Jun 2018, 2:17:11 am	⋮

Figure 9-7 Selecting data assets

Figure 9-8 shows the window where you can rename the columns.


IBM Watson Projects Tools Catalog Community Services Docs Support Manage

My Projects / Auto Insurance Claims / AutoClaims.csv / Data Refinery

+ Operation *Code an operation to cleanse and shape your data*

Data Profile Visualizations Steps

	olicy tring	Claim_Reason String	Sales Channel String	Total Claim String
1	orporate L3	Collision	Agent	384.81114
2	ersonal L3	Scratch/Dent	Agent	1131.4649
3	ersonal L3	Collision	Agent	566.47224
4	orporate L2	Collision	Call Center	529.88134
5	ersonal L1	Collision	Agent	138.13087
6	ersonal L3	Hail	Web	159.38304
7	orporate L3	Collision	Agent	321.6
8	orporate L3	Collision	Agent	363.02968
9	orporate L3	Collision	Agent	511.2
10	pecial L2	Hail	Branch	425.52783

1 STEPS

Data Source : AutoClaims....

Rename column JUST ADDED

Renamed column State Code to State_Code

Details Help

DATA FLOW

AutoClaims.csv_flow

Enter a description of the data flow

Steps: 1

Project: Auto Insurance Claims

DATA FLOW OUTPUT

Location: Auto Insurance Claims/Data ...

DATA SET NAME

AutoClaims.csv_shaped...

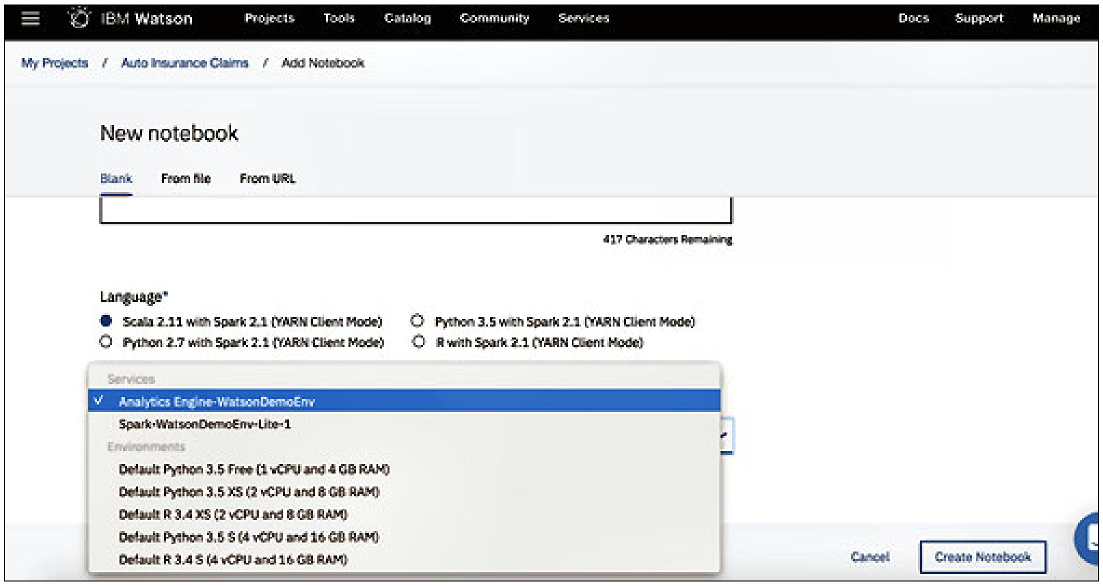
More

Display a menu SOURCE FILE: AutoClaims.csv SAMPLE SIZE: First 1000 rows

Figure 9-8 Data Refinery window

Create a notebook

For this notebook, associate the Analytics Engine instance (which was associated with the project earlier) with the run time. This association supports different kinds of kernels such as the Scala, Python, and R. Therefore, if you have associated different kinds of run times at the project level, you can choose between them at the notebook level, as shown in Figure 9-9.



New notebook

417 Characters Remaining

Language*
● Scala 2.11 with Spark 2.1 (YARN Client Mode)
○ Python 3.5 with Spark 2.1 (YARN Client Mode)
○ Python 2.7 with Spark 2.1 (YARN Client Mode)
○ R with Spark 2.1 (YARN Client Mode)

Services
✓ Analytics Engine-WatsonDemoEnv
Spark-WatsonDemoEnv-Lite-1
Environments
Default Python 3.5 Free (1 vCPU and 4 GB RAM)
Default Python 3.5 XS (2 vCPU and 8 GB RAM)
Default R 3.4 XS (2 vCPU and 8 GB RAM)
Default Python 3.5 S (4 vCPU and 16 GB RAM)
Default R 3.4 S (4 vCPU and 16 GB RAM)

Figure 9-9 Selecting Analytics Engine-WatsonDemoEnv

Insert To Code utility

Insert To Code is a utility that abstracts much of the connectivity details to the data asset on object store. It automatically detects the language/kernel of the notebook and offers appropriate code templates that you can choose from. It also automatically discovers the object store credentials that are needed for connection and stores those details as well. Click the **Insert to Code** icon as shown in Figure 9-10 to use this feature. This example uses code that uses Spark sessions and data frames.

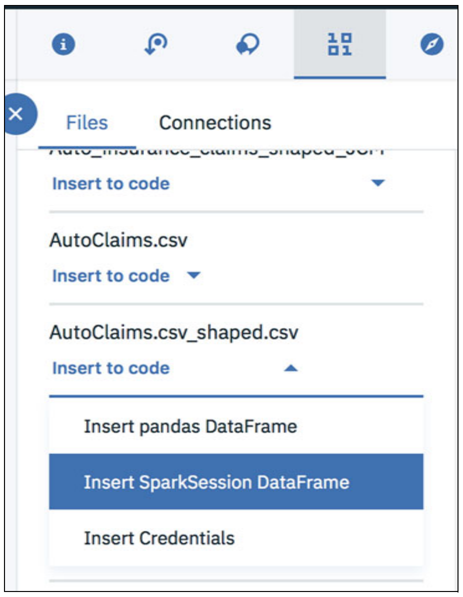


Figure 9-10 Insert SparkSession DataFrame

Example 9-1 shows the code that is generated.

Example 9-1 Example code using Spark sessions and dataframes

```
In [10]:
import ibmos2spark

# @hidden_cell
credentials = {
    'endpoint': 'https://s3-api.us-geo.objectstorage.service.networklayer.com',
    'api_key': '<MASKED>',
    'service_id': 'iam-ServiceId-2929d336-cff7-45d2-b621-0b0f4a6ac4a2',
    'iam_service_endpoint': 'https://iam.ng.bluemix.net/oidc/token'}

configuration_name = 'os_0c9d5e40ed09450e8f6a62d73321f98f_configs'
cos = ibmos2spark.CloudObjectStorage(sc, credentials, configuration_name,
'bluemix_cos')

from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
df_data_1 = spark.read\
  .format('org.apache.spark.sql.execution.datasources.csv.CSVFileFormat')\
  .option('header', 'true')\
  .load(cos.url('data_asset/AutoClaims.csv_shaped_WfLT3mxrREqJEb7yBbFkmA.csv',
'autoinsuranceclaims84fafb4d8a464df4bfc813eaf52415d3'))
```

Use Spark SQL along with plotting library to get a simple chart

The auto generated code in Example 9-1 reads the data from IBM Cloud Object Storage and creates a dataframe. To run SQL statements using Spark, register the data as a temporary table:

```
df_data_1.registerTempTable("auto_claims")
```

Use the plotting library `matplotlib` to create a simple bar chart, as shown in Example 9-2.

Example 9-2 Using the matplotlib plotting library to generate a chart

```
%matplotlib inline
import matplotlib.pyplot as plt, numpy as np
pandaPlot = sqlContext.sql("select CLAIM_REASON, count(*) AS COUNT from
auto_claims GROUP BY CLAIM_REASON").toPandas()


pandaPlot.plot(kind='bar', x='CLAIM_REASON', y='COUNT', figsize=(12, 5))
```

The graph in Figure 9-11 shows that claims due to collision are the most numerous followed by natural forces such as hail, and then reasons such as a scratch/dent.

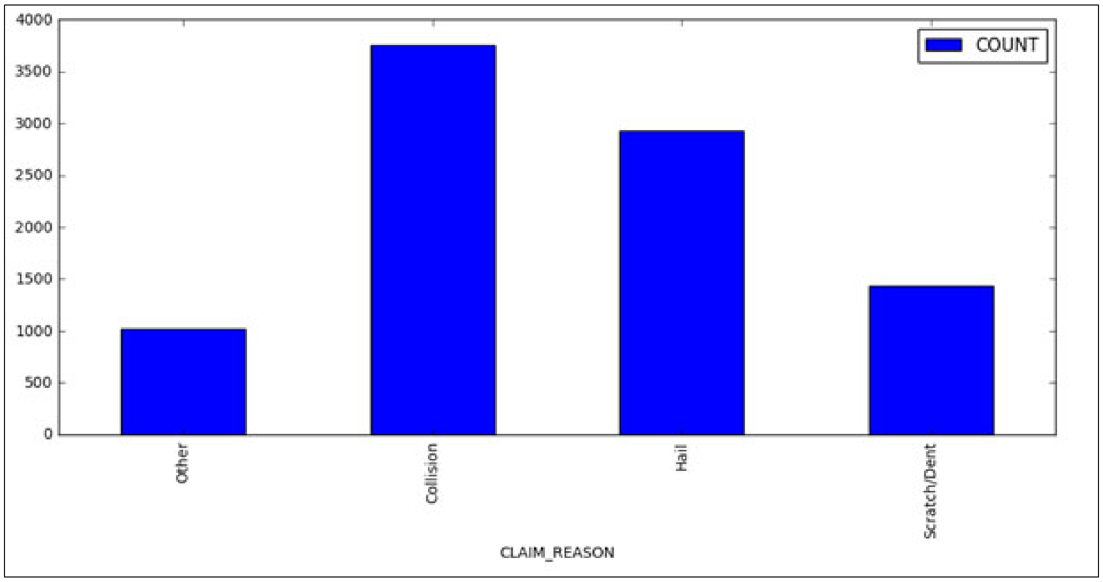


Figure 9-11 Claim reason chart

The notebook for reference can be accessed from the Module 9 resources in GitHub. This is a very simple example, but you can use the full power of Spark-based analytics for different use cases.

9.6 Next steps

Now you have successfully enhanced the Watson Visual Recognition service for your application. Many other use-cases using this service can still be added, so do not hesitate to experiment further.

9.7 Other references

- *Building Cognitive Applications with IBM Watson Services: Volume 1 Getting Started*, SG24-8387
- *Building Cognitive Applications with IBM Watson Services: Volume 3 Visual Recognition*, SG24-8393

10 Adding geographic and weather data

Welcome to Module 10 in this tutorial series! This series explains how to build a cloud-native, consumer-facing car insurance claim application on IBM Cloud that is built according to best practices, uses IBM Cloud Object Storage at its core, and employs AI and machine learning to provide a unique digital experience for a customer.

This module teaches how to extract metadata from images of automobiles and use that data to look up weather data in the area where the pictures were taken. This module assumes that all nine previous modules were followed to completion.

Insurance companies need a way to battle fraud, and in this scenario, Insurance Company X uses the image and weather data to help detect fraud. By taking the data and location information from an image, the insurance company can perform these tasks:

- Check to see whether the image submitted actually occurred in the expected time frame.
- Compare the collected weather data with what was reported in the claim. If the weather does not match what was submitted with the claim, then it can be rejected and reported as fraud.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

10.1 Learning objectives

After completing this module, you will be able to:

- Create and configure an instance of the Weather Company Data service.
- Implement code to extract location and time from an image's EXIF metadata.
- Using the extracted metadata, implement code to get the Weather Data for the location and time of the image.
- Identify how to use the Weather Data to detect insurance fraud.

10.2 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 9 to get the finished code from the previous module that you will build on:

```
$ cd module09
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module10
```

10.3 Implementation

The Weather Company® Data APIs allow the user to access weather data for a specified geolocation. This module uses the Historical Data API, which returns up to 24-hour historical weather observations based on a geolocation extracted from the submitted images.

When pictures are taken from a digital camera, certain details are stored with the image file. This data is called the Exchangeable Image File Format (Exif) metadata. The Exif data stores information like the date and time that the photo was taken and its geolocation. This data is used in the call to the Historical Data API to get weather data for the date, time, and location of the accident.

In Module 4, the Weather Data Service instance was created and bound to the application starter kit. Complete the following steps if you would like to create your own service instance:

1. Create a Weather Company Data instance within IBM Cloud. Open the IBM Cloud dashboard, select **Catalog**, select **Platform** → **Data & Analytics**, and click **Weather Company Data**. You can also enter `weather` in the **Search** textbox and the choices will be filtered automatically.
2. Name the service. You can use the default or change the name, and then click **Create**.

> **Note:** This module uses the Free Plan that allows up to 10,000 API calls, which is plenty for the purpose of this tutorial.

On the Weather Data console, you can read more about the service by following the links under the Get Started section to access the service documentation.

3. Click the **Menu** icon and select **Web Apps**.
4. Click **Apps** and then click **Your Starter Kit App**.
5. Click **Add Resource** and select **Existing Services**, then click **Next**.
6. Select the Weather Company Data service created earlier in the module and click **Next**.

   The service is now bound to the application and the credentials should be displayed in the **Credentials** text box.
7. If you would like to create the Service credentials separately before binding, from the Weather Company Data instance console, click **Service credentials**, as shown in Figure 10-1.

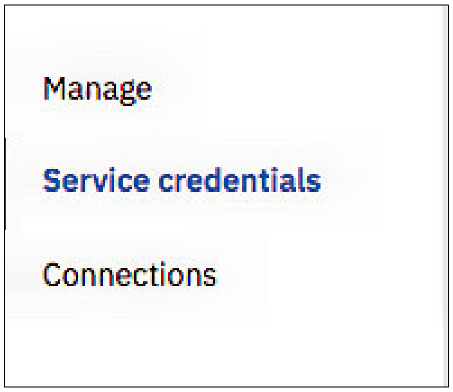
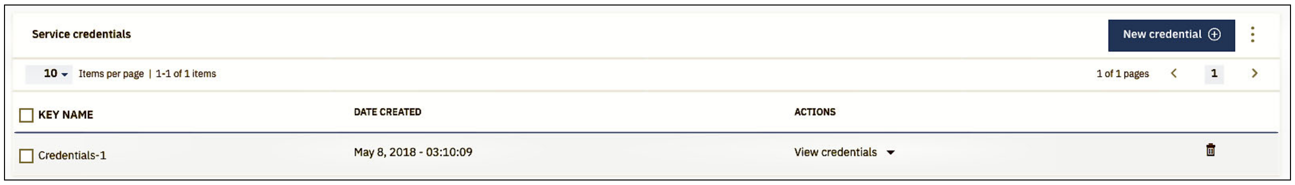


Figure 10-1 Selecting Service credentials

8. Click the **New credential** button.
9. Name the credential or accept the default. For this module, you will not need to add any optional Inline Configuration Parameters. Click **Add**.

   The new credential is now listed in the Service credentials table, as shown in Figure 10-2.



Service credentials

New credential

10 Items per page | 1-1 of 1 items 1 of 1 pages 1

KEY NAME	DATE CREATED	ACTIONS
Credentials-1	May 8, 2018 - 03:10:09	View credentials

Figure 10-2 Service credentials table

10. Click the twistie beside **View credentials** to see the generated JSON, as shown in Example 10-1. Note that in the example, the username and password have been removed.

Example 10-1 Generated JSON

```
{
  "username": "<USERNAME>",
  "password": "<PASSWORD>",
  "host": "twcservice.mybluemix.net",
  "port": 443,
  "url": "https://USERNAME:PASSWORD@twcservice.mybluemix.net"
}
```

Now, when you bind the service instance to your applications, the service credentials you created will be used instead of being newly generated.

11. When the code is downloaded, the `/server/config/mappings.json` file is provisioned with the Weather Insights® mapping and the credentials are added to `/server/localdev-config.json`.

    Example 10-2 shows the contents of the `mappings.json` file.

Example 10-2 Contents of the mappings.json file

```
"weather_company_data_url": {
    "searchPatterns": ["cloudfoundry:$.weatherinsights[0].credentials.url",
"env:service_weather_company_data:$.url",
"file:/server/localdev-config.json:$.weather_company_data_url"]
}, "weather_company_data_username": {
    "searchPatterns":
["cloudfoundry:$.weatherinsights[0].credentials.username",
"env:service_weather_company_data:$.username",
"file:/server/localdev-config.json:$.weather_company_data_username"]
}, "weather_company_data_password": {
    "searchPatterns":
["cloudfoundry:$.weatherinsights[0].credentials.password",
"env:service_weather_company_data:$.password",
"file:/server/localdev-config.json:$.weather_company_data_password"]
}
```

Example 10-3 shows the contents of the `localdev-config.json` file.

Example 10-3 Contents of the localdev-config.json file

```
"weather_company_data_url":
"https://<USERNAME>:<PASSWORD>@twcservice.mybluemix.net",
"weather_company_data_username": "<USERNAME>",
"weather_company_data_password": "<PASSWORD>"
```

Note: In the `localdev.config.json` file, the username and password have been removed and replaced with variables. In your file, the actual credentials will be displayed.

12. You have provisioned a new instance of the Weather Company Data service. To extract the time and geolocation from the image and convert the data into usable coordinates, install the `imagemagick` and `dms2dec` modules:

```
$ npm install imagemagick
$ npm install dms2dec
```

13. These two modules must also be added to the dependencies section in `insurancewebappbackend/package.json`.

14. Add the following two lines to `package.json`:

```
"dms2dec": "^1.1.0",
"imagemagick": "^0.1.3"
```

15. Create a folder in `server/services` called `weather` and create a file in that directory called `weather-data.js`:

```
$ cd server/services && mkdir weather && touch weather-data.js
```

16. Now that the `weather-data.js` file is created, look at the service code that makes a request to the Weather Company Data API and returns the weather observations for the past 24 hours (Example 10-4).

Example 10-4 Code that returns weather observations

```
const IBMCloudEnv = require("ibm-cloud-env");
const request = require("request");
const log4js = require("log4js");
const logger = log4js.getLogger("weather-service");

//Public methods
module.exports.getWeatherSituation = _getWeatherSituation;

// Private methods
function _getWeatherSituation(lat, lon) {
    return new Promise(function(resolve, reject) {
        const config = {
            username: 
IBMCloudEnv.getString("weather_company_data_username"),
            password: 
IBMCloudEnv.getString("weather_company_data_password"),
            url: IBMCloudEnv.getString("weather_company_data_url"),
            hours: "23"
        };
        let url = 
`${config.url}:443/api/weather/v1/geocode/${lat}/${lon}/observations/timeseries
.json?units=e&hours=${config.hours}`;
        request(url, function(err, response, body) {
            if (err) {
                logger.info("error:", err);
                reject(err);
            } else {
               logger.info("statusCode:", response && response.statusCode);
                resolve(JSON.parse(body));
            }
        });
    });
}
```

The function that you want to examine is `_getWeatherSituation(lat, lon)`. The function starts by building the config that consists of a username, password, url, and hours, as shown in Example 10-5.

Example 10-5 _getWeatherSituation(lat, lon) function

```
const config = {
    username: IBMCloudEnv.getString("weather_company_data_username"),
    password: IBMCloudEnv.getString("weather_company_data_password"),
    url: IBMCloudEnv.getString("weather_company_data_url"),
    hours: "23"
};
```

The credentials used were created when the service was bound to the application and stored in `localdev-config.json`. The `hours` represent how many hours to go back. Setting `hours` to `23` tells the weather service to get the weather observations for the current hour and then get the past 23 hours of historical data. This process results in 24 hours' worth of historical weather data. The next step, the `config`, is used to build the `url` that will be used in the HTTP request.

```
let url = `${config.url}:443/api/weather/v1/geocode/${lat}/${lon}/observations/timeseries.json?units=e&hours=${config.hours}`;
```

The `config.url` file already contains the username, password, and url, but you might be wondering what `&{lat}/${lon}` mean. These variables will contain the latitude and longitude values pulled from the Exif metadata. The code to obtain these values is covered later in the module. The last variable in the `url` is `config.hours`, which specifies how many hours to retrieve.

The endpoint (or API) in the example is `/observations/timeseries.json`. The `?units=e` parameter specifies what unit of measure to return. This tutorial uses `e` for Imperial (English). The other units that are supported are `m` for metric and `h` for hybrid.

Now that the code has built the `url`, the HTTP request is made, and the results are returned, as shown in Example 10-6.

Example 10-6 Results of HTTP request

```
request(url, function(err, response, body) {
            if (err) {
                logger.info("error:", err);
                reject(err);
            } else {
               logger.info("statusCode:", response && response.statusCode);
                resolve(JSON.parse(body));
            }
        });
```

The `request(url, function(err, response, body)` line makes the request and then error checking makes sure it was successful. If not, the error is returned and logged, and the request is rejected. If it was successful, the code logs the status code and returns the body of weather observations.

At this point, the `weather-data` class exits and control is returned to the calling function.

17. Now that the service code is complete, modify several of the existing files to fully implement the service in the application. Open `server/routes/claims/postClaimImage.js`.

18. At the top of the file in the list of constants, add the following line:

    ```
    const weather = require("../../services/weather/weather-data");
    ```

    This change allows the `postClaimImage.js` file to access the `weather-data.js` file that you created earlier in the module.

19. Search the file for `return imageProcessor.getMetadata(file.path);`. After the proceeding`})`, press **Return** to add a line and insert the code shown in Example 10-7.

Example 10-7 Code to add after return imageProcessor.getMetadata(file.path);

```
.then(metadata => { // some images don't contain exif metadata, so let's check
    if (metadata.exif) {
        logger.info("Extracted image metadata");
```

```
        image.imageMetadata = metadata.exif; // get the lat,lon from the image metadata
        const latLon = imageProcessor.getLatLon(image.imageMetadata);
        if (!latLon) {
            return;
        }
        logger.info("exif lat,lon", latLon);
        logger.info(`getting weather situation for lat:${latLon[0]} lon:${latLon[1]} @ time: ${image.imageMetadata.dateTimeOriginal}`);
        return weather.getWeatherSituation(latLon[0], latLon[1]);
    } // we don't have any metadata to use for weather so just return;
    logger.warn("No image metadata, skipping Weather API");
    return;
}).then((weatherSituation) => {
    if (weatherSituation) {
        image.weatherData = getClosestWeatherObservation(image.imageMetadata.dateTimeOriginal, weatherSituation);
    } // resize the image to get a normalized thumbnail
    return imageProcessor.resize(file.path);
})
```

The code in Example 10-7 is run when the image is being processed after the upload.

20. Add the function shown in Example 10-8 to the `postClaimImage.js` file.

Example 10-8 Function to add to postClaimImage.js

```
// Gets the weather observation in weatherSituation that is closest to the
// dateTime provided
function getClosestWeatherObservation(dateTime, weatherSituation) {
    const imageTime = dateTime / 1000; // sort observations by their distance to image time
    const closestObservation = _.sortBy(weatherSituation.observations, (observation) => Math.abs(imageTime - observation.valid_time_gmt)); // Weather api only returns data from last 24h. // A correct implementation would not add data if the image is outside of that window // for the sake of this demo, we're still adding the weather closest to the time // the picture has been taken
    logger.info("Weather:" + closestObservation[0]);
    return closestObservation[0];
}
```

21. In a previous step, you added the line "`const latLon = imageProcessor.getLatLon(image.imageMetadata);`" This line calls the function `getLatLon(metadata)` in the `processor.js` file. Open the `/server/services/image-processing/processor.js` file.

22. Add the code shown in Example 10-9 to the end of the file.

Example 10-9 Code to add to /server/services/image-processing/processor.js

```
function _getLatLon(metadata) {
    var gpsLatitude = metadata.gpsLatitude;
    var gpsLatitudeRef = metadata.gpsLatitudeRef;
    var gpsLongitude = metadata.gpsLongitude;
    var gpsLongitudeRef = metadata.gpsLongitudeRef;
    if (gpsLatitude && gpsLongitude && gpsLongitudeRef && gpsLatitudeRef) {
        return dms2dec(gpsLatitude, gpsLatitudeRef, gpsLongitude, gpsLongitudeRef);
    }
}
```

The `getLatLon(metadata)` function retrieves the latitude and longitude of the image data and processes it to decimal form.

You have added the necessary backend code, but that does you no good until the data can be displayed to the customer.

23. Open `/public/js/actions.js`. Search for the function `renderImageDetails(imageMetadata, weatherDetails, vrDetails)`. You will see the code shown in Example 10-10.

Example 10-10 Code in /public/js/actions.js

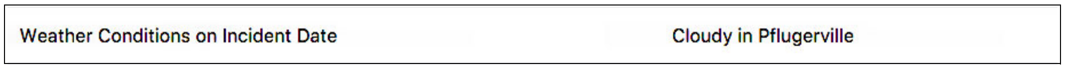
```
table_content += "<tr>";
table_content += "<td>Damage Type</td>";
table_content += "<td>Bump on Fender</td>";
table_content += "</tr>"
```

24. After the line `table_content += "</tr>"`, insert the code shown in Example 10-11.

Example 10-11 Code to insert into /public/js/actions.js

```
table_content += "<tr>";
table_content += "<td>Weather Conditions on Incident Date</td>";
if (weatherDetails) {
    table_content += "<td>" + weatherDetails.wx_phrase + " in " + weatherDetails.obs_name + "</td>";
} else {
    table_content += "<td>Weather not available.</td>";
}
table_content += "</tr>"
```

These lines of code add the weather conditions to the Image Details user-interface, as shown in Figure 10-3.



Weather Conditions on Incident Date	Cloudy in Pflugerville

Figure 10-3 Weather Conditions display

If there are no weather conditions available, the message “Weather not available.” is displayed in the table.

25. Now that the code is in place, test the updates by navigating to `/scripts` and creating a file called `datefaker.sh`:

```
$ cd scripts/ && touch datefaker.sh
```

26. Edit `datefaker.sh` and add the code shown in Example 10-12.

Example 10-12 Code for datefaker.sh

```
#!/bin/sh

# This script replaces exif metadata in a photo to replace the dateTimeOriginal value with
# the current time according to the system clock. This is to allow us to pull weather data
# from the Weather Company Data API which only provides the previous 24 hours

# Usage: place in a directory with some images where you want to fake the date

#Check if the tool is installed
command -v exiftool >/dev/null 2>&1 || { echo >&2 "I require exiftool but it's not installed. Download at https://www.sno.phy.queensu.ca/~phil/exiftool/ Aborting."; exit 1; }

#Check if you are in the right directory
if [ -n "$(ls -A ./*.JPG 2>/dev/null)" ]
then
  now=$(date +"%Y:%m:%d %T")
  exiftool -DateTimeOriginal="$now" "./"
else
  echo "No JPG images found, please copy the script to the images directory. Aborting."
  exit 1
fi
```

This script looks for a module called `exiftool`. This module must be installed separately at `https://www.sno.phy.queensu.ca/~phil/exiftool/`. Select the correct installation for your OS.

27. Set `datefaker.sh` as an executable file:

    ```
    $ chmod +x datefaker.sh
    ```

28. Navigate to the `/testData` directory. This directory contains four images: The front, back, left, and right sides of a car. Because the metadata needs to update to get weather data from the last 24 hours, issue the following command:

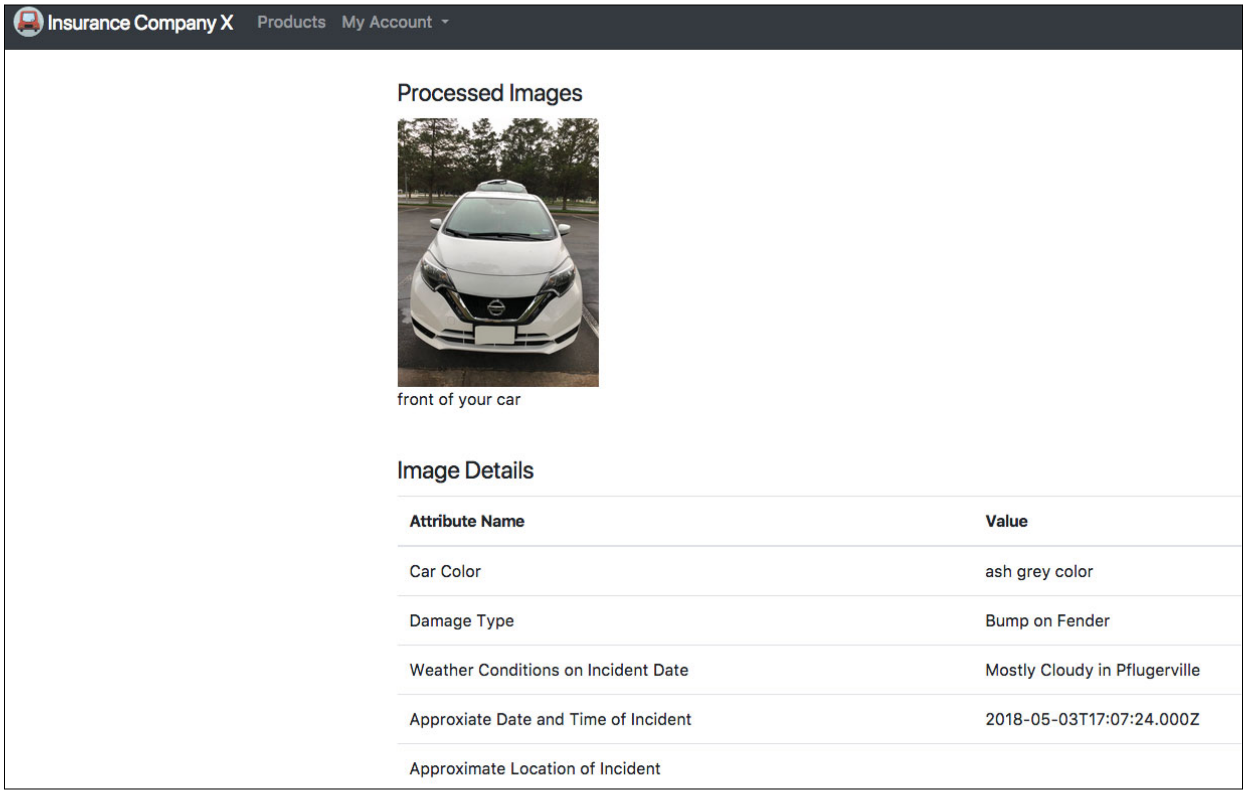
    ```
    $ ../scripts/datefaker.sh
    ```

    You should get the following output:

    ```
        1 directories scanned
        4 image files updated
    ```

29. Start the application locally by issuing the **`npm start`** command.
30. Open a browser and point it to `localhost:3000`.
31. Click **Login** and then either log in using existing credentials or create new credentials.
32. Click **Make a Claim**.
33. Under Upload Images, click **Browse**, navigate to the testData directory, select the file CAR_TEST_1_OR_FRONT.JPG, and click **Open**.

34. Click **Upload**. The file is uploaded as shown in Figure 10-4.



Attribute Name	Value
Car Color	ash grey color
Damage Type	Bump on Fender
Weather Conditions on Incident Date	Mostly Cloudy in Pflugerville
Approxiate Date and Time of Incident	2018-05-03T17:07:24.000Z
Approximate Location of Incident	

Figure 10-4 Uploading the image

10.4 Next steps

Review what you have learned and move on to Chapter 11, "Managing the application's users" on page 177.

10.5 Other references

- The Weather Company Data Documentation

  https://console.bluemix.net/docs/services/Weather/index.html
- Imagemagik

  https://www.npmjs.com/package/imagemagick
- Dms2dec

  https://www.npmjs.com/package/dms2dec
- Exiftool

  https://www.sno.phy.queensu.ca/~phil/exiftool/

Managing the application’s users

In the previous modules in this series, you have seen how to build a fully functional application that allows users to create claims and attach images. The application uses several IBM Cloud services to add intelligence to this process. However, the application currently only allows for a single user. There is no account concept and users cannot log in to get access to their own claim information.

In this module, you will add an account model to your system and integrate with IBM Cloud App ID to allow users to sign in using an account that they create or using a public social profile such as Google or Facebook. This configuration gives users the flexibility to log in how they choose, and makes new account creation easier (Figure 11-1).

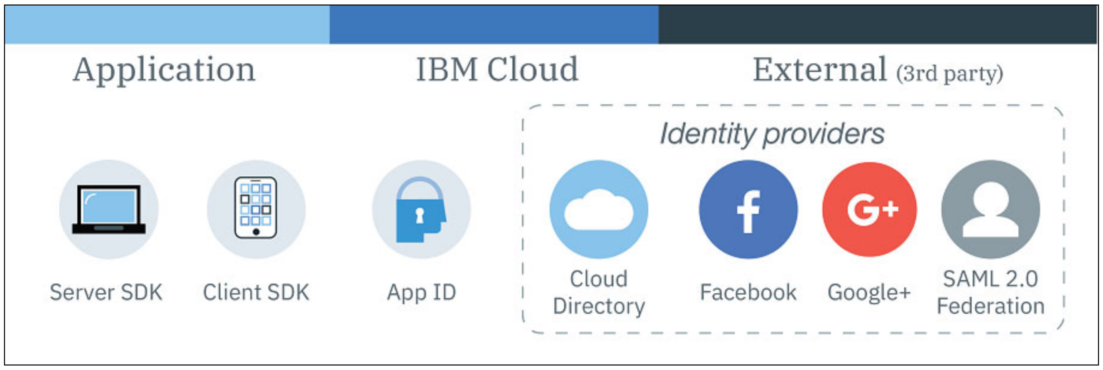


Figure 11-1 IBM Cloud structure

The App ID service provides uniform authentication and identify management capabilities for your applications. It provides a single API and callback mechanism to allow single sign-on for web applications. It also provides Cloud Directory, which is a user repository that allows your app users to sign up and sign in with email and password, verify email addresses, and manage password change and reset support. App ID includes support for Google and Facebook social identity providers. Using App ID makes it easy for you to add all of these authentication mechanisms with no additional development time. New providers can be added in the future without code modifications.

App ID also provides support for custom user attributes, allowing you to gather additional information about users. SAML for enterprise authentication is also supported. This module does not cover either function, but more information can be found in the App ID documentation.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Configuration
- Implementation
- Next steps
- Other references

11.1 Learning objectives

After completing this module, you will be able to:

- Create and configure an instance of App ID.
- Design a data model for users and accounts to provide authentication and access control functions.
- Implement authentication and access control.

11.2 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 10 to get the finished code from the previous module that you will build on:

```
$ cd module10
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module11
```

11.2.1 Basic concepts

Before adding App ID to your application, it is important to understand a few key principles of user management. Correctly designing your application to deal with identity can provide a more positive experience for your users and provide much needed flexibility for advanced features (Figure 11-2).

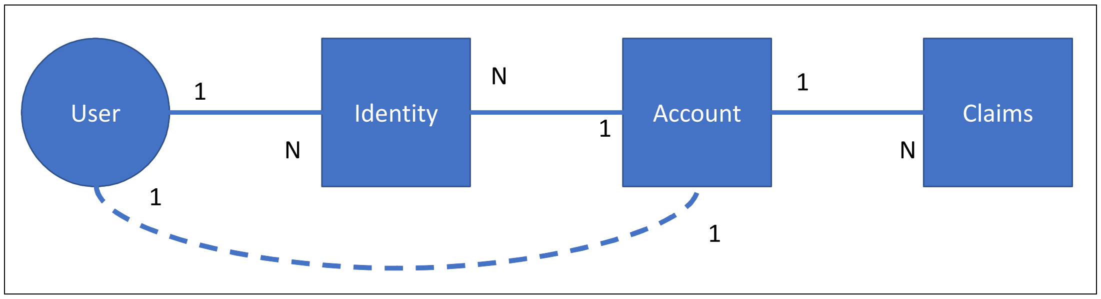


Figure 11-2 Identity loop

Your users will be most familiar with these concepts:

- User: Your app user will typically be a human who will sign up to use your app. We will be making our app map users into something meaningful so that the user can only view and edit their own claims.
- Credentials: A set of secret information that a user possesses that allows them to authenticate themselves. Because we are using App ID, our app will not have to directly manage credentials. This is one of the key benefits of App ID.

From your perspective as an app developer, it is important to understand the difference between these concepts:

- Identity: An identity maps a user to a set of credentials. The act of authentication has a user provide credentials to prove that they have some identity. Identity is not managed by the example app, so it is provided when a user authenticates using App ID.
- Account: An account is something that maps elements in your app's data model to a user. In the app, an account allows you to perform authorization to allow a user to see specific claims.

What is the difference between identity and account, and why do we need both concepts?

If you remember only one thing about identity, it should be this: Users often have more than one. They can sign up for your app using email and password, or by using Google or Facebook authentication. Users can have any combination of these types of accounts.

A well-designed app will add an account concept to the data model independent of identity. This configuration makes it possible for users to associate more than one identity to an account, providing them flexibility in how they log in. A user might want to use Google today, but switch to an email address and password later, for example. Or new third-party identity providers might be introduced in the future that you want to allow existing users to use. This change is fairly simple to do if it is designed in from the outset, so we will do so here.

11.2.2 Prerequisites

If you have completed the previous modules, you are ready to add App ID support.

However, note that when you create an instance of App ID, it uses the App ID Default Configuration for Facebook and Google providers. This configuration only allows you to authenticate up to 100 times per day. For production use, or for more heavy use during development, you must register with those providers to obtain an app-specific ID and secret. In the App ID console, you can get more information by clicking **App ID Console** → **Identity Providers** → **Manage**.

11.3 Configuration

If you have completed Module 4 and created a starter kit with all of the services used for this application, you might already have an App ID service instance created, and service credentials that are already added to your application service instance, starter kit code, and local development environment.

If you did not complete that step, add an instance to your web app after it is created. See Chapter 4, “Scaffolding using a Starter Kit” on page 29 for more details about how to add a resource to an existing project.

Ensure that the `server/services/` folder contains the `service-appid.js` boilerplate. You will be modifying that file to configure App ID.

For local development, add the following property to your `server/localdev-config.json` file:

```
"appid_app_external_address" : "http://localhost:3000",
```

For deployment, ensure that this property is set with the actual external host name and port for your app. See Chapter 4, “Scaffolding using a Starter Kit” on page 29 for more information on how to find this information.

11.4 Implementation

Now that you understand some basic concepts, this section covers what App ID actually does. When users authenticate with social identity providers, those providers allow users to authenticate and provide identity information to third-party sites using the OAuth2 grant flow. Figure 11-3 shows a sample work flow diagram.

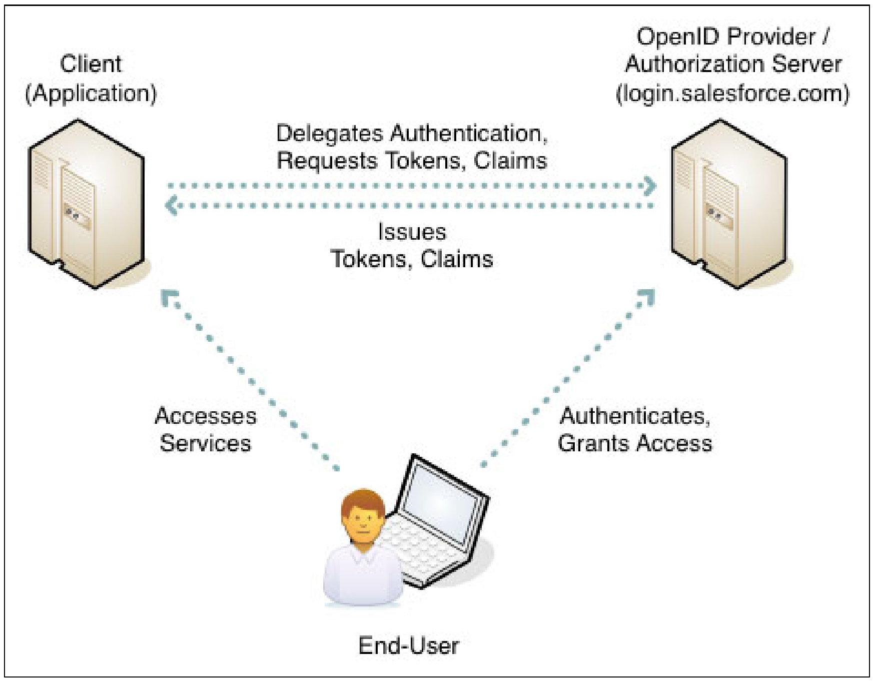


Figure 11-3 Work flow diagram

You can choose from a few different flows based on what your application is trying to do. In this module, because we are focusing on a web app that runs in a user's browser, we use the *web app strategy* using HTTP sessions. Figure 11-4 shows a sample diagram.

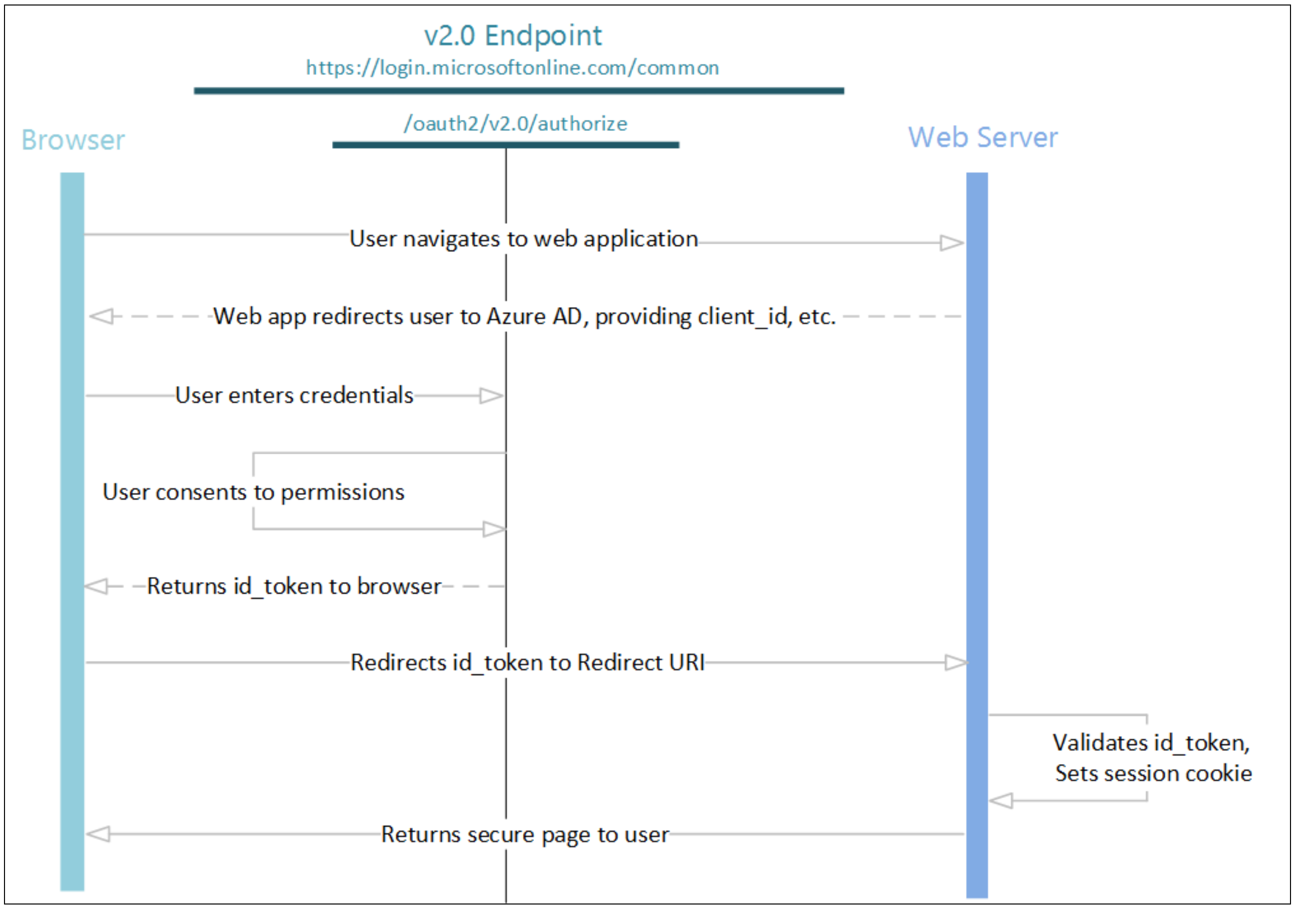

v2.0 Endpoint
https://login.microsoftonline.com/common
/oauth2/v2.0/authorize
Browser
Web Server
User navigates to web application
Web app redirects user to Azure AD, providing client_id, etc.
User enters credentials
User consents to permissions
Returns id_token to browser
Redirects id_token to Redirect URI
Validates id_token,
Sets session cookie
Returns secure page to user

Figure 11-4 Process flow diagram

The web app strategy abstracts the OAuth2/OIDC `authorization_code` grant flow, making it easy for developers to integrate authentication into their web apps. Several steps are involved when the web app strategy is used:

1. A user's browser is redirected to App ID.
2. App ID redirects the browser to the identity provider sign-in page to enter user credentials.
3. After successfully authenticating with the identity provider, the user is redirected back to App ID.
4. A grant code is sent to the web app, representing the user's authorization for App ID to obtain the user's identity information.
5. The web app, using App ID server-side SDK, retrieves the user's access and identity tokens from App ID, storing them in the HTTP session.

We have chosen the web app strategy with HTTP session storage because it is relatively simple to implement. However, also consider using the API strategy with JSON Web Tokens (JWT), which App ID also supports. In this model, the tokens are held by the browser and sent during each request in the Authorization header using the Bearer schema.

```
Authorization: Bearer <token>
```

This technique allows authentication to be stateless because it eliminates the need for backend server-side storage for session cookies. This configuration provides scalability and security benefits, as well as providing a better model for truly stateless, RESTful services. See JSON Web Tokens vs. Session Cookies for a detailed comparison of the two approaches. App ID also supports this schema, which can be implemented on the server side with the API Strategy. You should always consider the API Strategy for REST APIs. It can easily be used for UI authentication as well, and works especially well for stateless applications, including applications created using IBM Cloud Functions.

11.4.1 Step-by-step

To integrate App ID functionality, you must complete these high-level steps:

1. Adjust the `service-appid.js` boilerplate to correctly configure App ID. Because you are using `Express.js`, configure `Passport.js`.
2. Create the user data model and create methods to store this data model in Cloudant.
3. Create a REST API to allow the UI to obtain user information.
4. Create an `Express.js` injector to make it easier to interact with account information in the controllers.
5. Add in passport authentication middleware and the user injector to REST API modules.
6. Create an index page to allow users to log in. Create a route for the users' home page after they are logged in.

Configuring boilerplate code

The boiler plate code that comes with the Express.js basic starter kit only loads some variables from the App ID instance. You must extend this code to properly configure authentication.

First, add some dependencies to your project. Complete these steps:

1. Because you are using the Web App Strategy with HTTP cookies, include session management. This application just uses memory sessions, but you should add in either Redis or a database backed session storage for a production application. Otherwise, your users must log in again every time their browser encounters a new container.

   ```
   $ npm install express-session
   ```

   Note: Session management is not required if JSON Web Token authentication is used. This tutorial does not cover that type of authentication in detail.

2. Install `passport` for authentication:

   ```
   $ npm install passport
   ```

3. Install `bluemix-appid`. This Node.js module is compatible with Passport.js and allows you to configure the authentication strategy that you will be using.

   ```
   $ npm install bluemix-appid
   ```

4. Add these dependencies to the top of your `service-appid.js` file. Some of these dependencies should already be provided by the boilerplate code. Ensure that you have the following imports:

   ```
   const passport = require("passport");
   const session = require("express-session");
   const WebAppStrategy = require("bluemix-appid").WebAppStrategy;
   ```

At this point, the boilerplate should have an API Strategy and Web App Strategy configured. We will not be using the API Strategy, but you can leave it in if you want.

Make the following modifications to the Web App Strategy:

1. Create a few constants for some of your redirects:

   ```
   const CALLBACK_URL = "/ibm/bluemix/appid/callback";
   const LOGIN_URL = "/ibm/bluemix/appid/login";
   ```

   The LOGIN_URL is where the browser starts the sign-in flow. The CALLBACK_URL is the endpoint where App ID will redirect the browser to when sign-in is complete.

2. What you really need to pass App ID is a *Redirect URI*. This URI is passed as a query parameter when you redirect to App ID, and needs to be a complete URL, including host name and port. For a development deployment, this can be the external IP address and port of your container. For production, it will most likely be a load balancer endpoint. Insert the code that loads this property using the IBMCloudEnv utility function for this:

   ```
   const APPID_REDIRECT_URI = IBMCloudEnv.getString("appid_app_external_address")
   + CALLBACK_URL;
   ```

3. Modify the WebAppStrategy constructor parameters to pass in the `APPID_REDIRECT_URI` directly.

4. Configure session management, and then initialize passport, configure it to use sessions, and add in your authentication strategies. You also need to configure user serialization/deserialization. Because we are not doing anything specific to our application here, this is mostly standard `Passport.js` boilerplate, as shown in Example 11-1.

Example 11-1 Passport.js

```
// Set up session and passport to be used by our routers

   // Note: You should use a persistent session store in practice, such as
Redis.
   app.use(session({
      secret: "1234",
      resave: false,
      saveUninitialized: true,
      cookie: {
         httpOnly: false,
         secure: false,
         maxAge: (4 * 60 * 60 * 1000)
      }
   }));

   app.use(passport.initialize());
   app.use(passport.session());

   passport.use(webStrategy);
   passport.use(apiStrategy);


   // Configure passportjs with user serialization/deserialization. This is
required
   // for authenticated session persistence accross HTTP requests. See
passportjs docs
   // for additional information http://passportjs.org/docs
   passport.serializeUser(function(user, cb) {
```

```
      cb(null, user);
   });

   passport.deserializeUser(function(obj, cb) {
      cb(null, obj);
   });
```

5. To finish off service configuration, configure your Callback URL to receive logged in users. Also, create a logout endpoint so that you can add a **Sign Out** button to the UI. The logout endpoint clears the session, removes the browser's cookies, and redirects to the index page that you will create later, as shown in Example 11-2.

   Example 11-2 Configuring the Callback URL

   ```
   app.get(CALLBACK_URL,
   passport.authenticate(serviceManager.get("appid-web-strategy-name"),
      {allowAnonymousLogin: true}));


   app.get("/logout", function(req, res) {
      //stringify(req);
      req.session.destroy(function() {
         res.clearCookie("connect.sid");
         res.redirect("/");
      });
   });
   ```

6. In the `server/services/index.js` file, make sure that this line to properly configure App ID on startup is included:

   ```
   require("./service-appid")(app, serviceManager);
   ```

Creating the data model

Now that the service is configured, add your user model, as shown in Example 11-3.

Example 11-3 Adding the user model

```
{
   "type" : "user",
   "userId" : "77a48681-61bb-4fa1-8807-e40a734977b3",
   "accountId" : null,
   "name" : "John Smith",
   "identity" : {
      "provider" : "google",
      "id" : "123-EXAMPLE-ID-ABC"
   }
}
```

Because this tutorial uses Cloudant as the application's database, store each user as a JSON document. You must be careful in how you perform updates to these documents. Design your schema to allow multiple identities to be associated with each account. Account for the fact that it is not possible to atomically insert a record. This configuration means that inserts will not fail if there is already a matching record, so your schema must account for that fact.

When a user logs in, the identity token that you receive during the authentication process provides two key values: The identity provider, and the ID specific to that provider.

This schema should meet these requirements:

- Map user identities to an account
- Map claims to users
- Allow a user to link multiple identities to the same account

We want to map user identities to accounts, and map accounts to claims. In an ACID compliant database, the most straightforward way to accomplish these goals would be to create a data model such as the one shown in Figure 11-5.

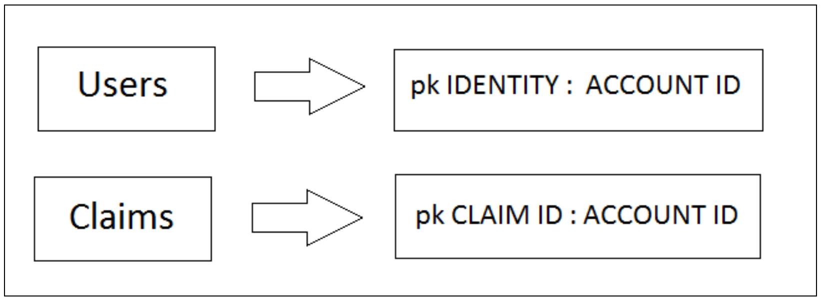


Figure 11-5 Simple data schema

For every identity encountered, add an entry in the table mapping the identity to an account ID. The account ID would then be mapped to a claim. If you want to link an identity to an existing account, you would atomically insert the identity with an existing account ID. If a user accidentally creates two accounts, Account A and Account B, and wanted to link them together, create a transaction to update Identity B to Account A and update the foreign key for all of the claims previously tied to Account B.

Be careful not to use the IDENTITY directly in the claim because this does not allow you to have multiple identities associated with an account.

To find claims for a logged in user, perform the following search:

1. Find the `accountId` for the current identity.
2. Select all claims where `accountId` matches.

However, there is a slightly better data model you can consider, as shown in Figure 11-6.

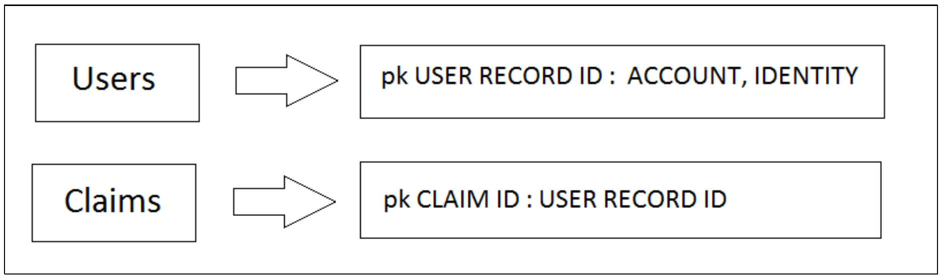

Users
pk USER RECORD ID : ACCOUNT, IDENTITY
Claims
pk CLAIM ID : USER RECORD ID

Figure 11-6 Improved data schema

This model is slightly better for our purposes because it allows you to link each claim to a record, and not have to update those claims later if you link identities. Instead, if you link an account later, simply add an account ID for the two user records. This action is especially important for an eventual consistency database like Cloudant because it is not possible to atomically update multiple records.

This limitation can result in inconsistent queries after an identity is linked if there are existing claims. Therefore, this is a better data model if you want to allow merging accounts. This is true for other databases as well because an atomic update for many claims can be quite expensive.

Additionally, it is better to structure changes as updates only and not require read-modify-writes. If you can simply update an account ID across two user records instead of looking up an account ID from one record and updating the other with it, it is safer. For example, when linking an account, how do you know that you are updating in the correct direction? This data model also lets you merge two existing accounts together cleanly.

With this model, perform the following search to find claims for a given user:

1. Find the `user_id`, and account ID (if it exists) for the current identity.
2. If the account id exists, find the set of user IDs that match that account.
3. Select all claims where user ID matches any of the set of user IDs from step 1 and 2.

In the end, both schemas can work. You should consider how much flexibility you want to allow for your users. When possible, allow users to link new identities to an existing account. Also consider if you want to allow merging existing accounts because this is a commonly requested user feature in many industries.

In our code, we have a placeholder for accountId, but will actually implement the first model. The claim search algorithm will only find a single user ID from an identity and find claims matching that ID. If you would like to extend this code to support merging claims or adding multiple identities, it is relatively easy to do.

To implement this data model, first create an index document called `userIndex.json`, as shown in Example 11-4, and place it in the `server/services/cloudant` folder.

Example 11-4 Code for userIndex.json

```
{
  "name": "userIndex",
  "ddoc": "userIndex",
  "type": "json",
  "index": {
   "fields": [
     {
       "type" : "asc"
     },
    {
      "userId" : "asc"
    },
    {
      "accountId" : "asc"
    },
    {
      "identity.provider" : "asc"
    },
    {
      "identity.id" : "asc"
    }
   ]
  }
}
```

Because we want to perform lookups by both user ID and identity, index both structures. Note that identity has two parts, provider and ID, so you must index both. This index also includes account ID, though we will not implement that part of the search algorithm in this code.

Make sure that this new index gets created in Cloudant by adding the following line near the top:

```
const USER_INDEX = require("./userIndex.json");
```

In the _init() method, add this code at the end to create the index:

```
.then(() => db.index(USER_INDEX)) // create the user index
.catch(e => {
logger.warn(e._data);
});
```

We are also going to be searching claims by `userId`, so add this line to `claimIndex.json` in the same folder:

```
{

        "userId": "asc"

    },
```

This code creates a new index in Cloudant with this additional field.

The `sort` parameter in `getClaim()` and `listAllClaims()` must be updated to reference this new index. Make sure that the sort parameter is in the exact same order as the new index you have created.

In `server/services/cloudant`, update `index.js` to add these methods to manage the user records:

```
_createUser()
_getUser()
_findUserByIdentity()
_findOrCreateUser()
_listUserClaims()
```

The first three methods create a new user, get a user by user ID, and find a user by identity. Because this simple application does not require a user to establish much of a profile before using the system, the fourth public method `_findOrCreateUser()` automatically creates a user record whenever a user authenticates to the system. A more advanced application might not automatically create a user, so we use these methods independently.

The final method you will create is a version of `listAllClaims()` seen in an earlier module. Change the query to restrict the returned claims to those for the current user.

The full code will have some other convenience methods dealing with users, but we will not use them in our application right now.

The actual code for finding and creating users is not much different from finding or creating a claim. To find a user, create a query with a selector and with a sort order matching the index that you have created. The selector is either the user ID or the identity. Return the first document if it exists. Note that with this schema, it is possible that a race condition during authentication could create two user records for the same provider. The second scheme, where the search algorithm finds a set of user records, would resolve this issue safely.

For the create user method, simply create a document, ensuring that the type is `user` and a new user ID is generated.

You can see these two methods in Example 11-5.

Example 11-5 User model methods

```
// This function creates a user.
function _createUser(userDetails) {
   // create the document from provided details, generated UUID and created date
   const document = Object.assign({},userDetails, {type: "user", userId: uuid(),
created: moment.utc()});
   // just in case someone tried to set the id
   delete document._id;
   // insert the document and return the claimId
   return db.insert(document).then((d) => {
      logger.debug(d);
      return document.userId;
   });
}

// This function finds a user by user id.
function _getUser(userId) {
   if (!userId) {
      return Promise.reject("Missing userId parameter");
   }
   // Leverage the index we created to sort by claimId and created date
   const query = {
      "selector": {
         "type" : "user",
         "userId": userId
      },
      "sort": [
         {
            "type" : "asc"
         },
         {
            "userId" : "asc"
         },
         {
            "accountId" : "asc"
         },
         {
            "identity.provider" : "asc"
         },
         {
            "identity.id" : "asc"
         }
      ]
   };

   // Search all documents for this claim
   return db.find(query).then((docs) =>  {
      // if there are none, return null
      if (!docs || docs.length === 0) {
         return null;
      }

      return docs.docs[0];
   });
}

// This function finds the set of users by an identity.
function _findUserByIdentity(provider, id) {
   if (!provider || !id) {
      return Promise.reject("Missing provider or id parameter");
   }
   // Leverage the index we created to sort by claimId and created date
   const query = {
      "selector": {
         "type" : "user",
         "identity.provider": provider,
         "identity.id" : id
      },
      "sort": [
         {
            "type" : "asc"
         },
         {
            "userId" : "asc"
         },
         {
            "accountId" : "asc"
         },
         {
            "identity.provider" : "asc"
         },
         {
            "identity.id" : "asc"
         }
      ]
   };

   // Search all documents for this claim
   return db.find(query).then((docs) =>  {
      // if there are none, return null
      if (!docs || docs.length === 0) {
         return null;
      }

      return docs.docs[0];
   });
}
```

The `findOrCreateUser()` method calls these three methods in turn. This process is straightforward and you can see this method in the full code sample download.

To create the `_listUserClaims()` method, copy the `_listAllClaims()` method, then change the selector to add in the user ID:

```
   "selector": {
      "type" : "claim",
      "userId": userId
   },
```

Finally, make sure to add exports at the top:

```
module.exports.listUserClaims = _listUserClaims;
module.exports.createUser = _createUser;
module.exports.getUser = _getUser;
module.exports.findUserByIdentity = _findUserByIdentity;
module.exports.findOrCreateUser = _findOrCreateUser;
```

Creating a REST API

The REST API for users will be relatively simple. Make a single endpoint, `/user`, that allows the UI to query the name of the currently logged in user.

Create the file shown in Example 11-6 as `server/routers/user.js`.

Example 11-6 The server/routers/user.js file

```
const express = require("express");
const passport = require("passport");
const serviceManager = require("../services/service-manager");
const common = require("./common");


module.exports = function(app) {
   var router = express.Router();

   // Get account information for the currently logged in user, creating
   // an account if it doesn't already exist.
   // Note: the injector does all the work here.
   router.get("/", function (req, res) {
      res.json(req.userAccount);
   });


   // Register the route /claim
   app.use("/user",
      passport.authenticate(serviceManager.get("appid-web-strategy-name")),
      common.cloudantInjector,
      common.userInjector,
      router);
};
```

This code returns a JSON representation of your user record. There is more going on here that we will examine more closely when you create your injector and insert passport authentication to the claims API.

Make sure that your new route gets registered by adding this line to `server/routers/index.js`:

```
require("./user")(app);
```

Creating an Express.js injector

In Module 5, you created a Cloudant injector, which is middleware to ensure that downstream code always has a reference to the Cloudant component.

We will do something similar for user records here.

In the `server/routers/common.js` file, add two methods:

- An internal method `_extractAuthIdentity()` to get the identity information for an authenticated request. This method depends on the `passport.authenticate()` middleware to insert the user identity information. Passport inserts information about all three token types. The token that we are interested in examining is the identity token, as shown in Example 11-7.

Example 11-7 Internal method

```
function _extractAuthIdentity(req) {
   // Passport inserts it's information into the key we've specified during
   // configuration, so we will retrieve that from the session here.
   var appIdAuthContext =
req.session[serviceManager.get("appid-web-auth-context")];

   if (appIdAuthContext) {
      // We're interested in the identitites stored in the identity token
payload.
      // For our application we assume there will only be a single identity per
session.
      if (appIdAuthContext.identityTokenPayload.identities.length > 0) {
         return appIdAuthContext.identityTokenPayload.identities[0];
      }
   }

   // If we have not logged in
   return null;
}
```

- Export the `userInjector` itself. This method first attempts to extract the identity information. It then calls the `findOrCreateUser()` method created above. See Example 11-8.

Example 11-8 Exporting the userInjector

```
module.exports.userInjector = function (req, res, next) {

   const identity = _extractAuthIdentity(req);
   if (!req.user) {
      // no user on request, proceed
      next();
      return;
   }
   req.cloudant.findOrCreateUser(req.user.name, identity.provider, identity.id)
      .then((userRecord) => {
         req.userRecord = userRecord;
         next();
         return;
      }).catch((err) => {
         logger.info(err);
```

```
        res.status(500).send(err);
     });
};
```

Adding REST API authentication

After all that, adding actual authentication for the REST API is easy. First, make sure that new claims get associated with the correct user record. You could do this when you create the initial claim, but in this example we add the `userId` when we add our first image. In `server/routers/claims/postClaimImage.js`, modify the `_processImage()` method to insert the `userId` right after the claimId, as shown in Example 11-9.

Example 11-9 The _processImage function

```
function _processImage(req,res) {
   …
   const append = {
      claimId: req.params.claimId,
      userId: req.userRecord.userId,
      images: [imageContainer]
   };
…
}
```

In `server/routers/claims/index.js`, modify the `/claim` route to only show claims for the current user by changing the call from `listAllClaims()` to `listUserClaims()`, as shown in Example 11-10.

Example 11-10 Changing the call to listUserClaims()

```
// GET a list of claims
router.get("/", function (req, res) {
   req.cloudant.listUserClaims(req.userRecord.userId)
.then((claims) => {
      res.json(claims);
   })
.catch((err) => {
      logger.error(err);
      res.status(500).send("Something broke!");
   });
});
```

Finally, just as seen in `user.js`, insert the passport authentication and user injector middleware in the `app.use()` call, as shown in Example 11-11.

Example 11-11 Adding passport authentication and user injector middleware

```
// Register the route /claim
app.use("/claim",
   passport.authenticate(serviceManager.get("appid-web-strategy-name")),
   bodyParser.json(),
   common.cloudantInjector,
   common.userInjector,
   router);
```

The passport authentication middleware forces authentication on these pages and the user injector middleware retrieves the user record.

Creating the final login page

As shown in Chapter 4, “Scaffolding using a Starter Kit” on page 29, the app has only two pages. The first, `index.html`, has a simple login page. Express routes to `index.html` by default. It is served using `express.static()`, and the user does not require authentication to view this page.

The second page is `home.html`, and it is served from the route `"/home"`. This is the location of the actual single page app. In `server/server.js`, change the registration for this route to require passport authentication. All that the **Login** button does is redirect to `"/home"`, and passport takes care of the rest.

First, import `passport`:

```
const passport = require("passport");
```

Then, update the registration of the `"/home"` route, as shown in Example 11-12.

Example 11-12 Registering the “/home” route

```
app.use("/home",
passport.authenticate(serviceManager.get("appid-web-strategy-name")),
(req, res) => {
      res.sendFile(path.join(__dirname, "../public", "home.html"));
});
```

11.5 Next steps

This module taught you how to add authentication and identity to your application easily using App ID. It also taught you how to build a simple data model that allows you to easily provide a flexible account model for your users. This data model deals well with non-transactional databases and also allows you to provide enhanced functionality to your users, such as linking accounts together when a user uses multiple social identity providers over time.

Note that this system simply limits viewing and editing claims to the account that created them. As you are designing your own applications, think about how you will perform more advanced access control.

Consider the following concerns, among others:

- Your user model must incorporate different types of users. This module covered a user who has a specific insurance policy. What about claims adjustors? System administrators? Design an account model that meets the different roles in your system.
- You must build an authorization model into your system. Can you break the actions that can be performed in the system into specific permissions? Which actions can users perform? A role-based system can allow you to flexibly change which actions that specific users can perform. You will also need to add permissions checks into different methods. For example, the API to retrieve an image does not currently check that the user should have access to that image, so it is vulnerable to attack if a user can guess what another image might be named.

- Most application developers will create a built-in authorization model. However, integration with SAML identity providers can allow organizations to manage roles on the identity provider. These roles can then be mapped to permissions in the application. App ID has additional tools to make working with SAML easier.

Review what you have learned and move on to Chapter 12, “Providing case search” on page 197.

11.6 Other references

- Learn more about App ID

  `https://www.ibm.com/cloud/app-id`
- More details about when to use the service

  `https://console.bluemix.net/docs/services/appid/about.html#about`
- User identity versus user account, and best practices for integrating third-party providers

  `https://cloudplatform.googleblog.com/2018/01/12-best-practices-for-user-account.html`
- JSON Web Tokens versus Session Cookies

  `https://ponyfoo.com/articles/json-web-tokens-vs-session-cookies`

12

Providing case search

Welcome to module 12 in this tutorial series! This series explains how to build a cloud-native, consumer-facing car insurance claim application on IBM Cloud that is built according to best practices, uses IBM Cloud Object Storage at its core, and employs AI and machine learning to provide a unique digital experience for a customer.

This module covers how to create and use search indexes within IBM Cloudant NoSQL DB. The search indexes are based on Apache Lucene full-text search and can be used to search database documents using the Lucene Query Parser Syntax. This module assumes that all 11 previous modules were followed to completion.

This section describes search indexes in theory, so it will not include any new code to add to the Insurance Company X application.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

12.1 Learning objectives

After completing this module, you will be able to:

- Define a search index
- Create search indexes
- Query search indexes

12.2 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 11 to get the finished code from the previous module that you will build on:

```
$ cd module11
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module12
```

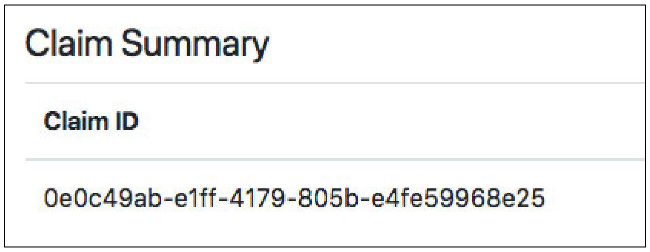
12.3 Implementation

Cloudant allows the user to create several different indexes to find the documents that contain the needed data. Those indexes are Primary, which is created automatically when a database is created, Secondary, Search, Geospatial, and Cloudant Query. As stated earlier, this module will focus on the Search Index.

The Search Index is built using Lucene search. Use it when ad hoc queries are needed on one or more fields, searches use large blocks of text, and queries require Lucene syntax like wildcards.

In the Insurance Company X application, the user can submit an insurance claim about damage to their automobile. When the **Make a Claim** button is clicked, a Claim ID is automatically generated. After that Claim ID is generated, a JSON document is inserted into the Cloudant database.

Figure 12-1 shows the claim used during the remainder of this tutorial.


Claim Summary

Claim ID

0e0c49ab-e1ff-4179-805b-e4fe59968e25

Figure 12-1 Example claim

The JSON document created in Cloudant is shown in Example 12-1.

Example 12-1 Created JSON document

```
{
    "_id": "a2141ef6e3f0bbdfb1482a569e807231",
    "_rev": "1-7cd1699cdf4819d294147f0747d85b28",
    "type": "claim",
    "claimId": "20916bf3-8d48-475a-944e-aacb76416a71",
    "created": "2018-05-18T16:55:25.914Z"
}
```

The data above does not tell you much, so you might never have a reason to use anything other than the document lookup or primary index. However, what happens to the JSON document when a single image is uploaded to the claim? The new JSON document is shown in Example 12-2.

Example 12-2 JSON document with uploaded image

```
{
    "_id": "7912f19a21e0b5c74b102e7fa314cc72",
    "_rev": "1-fd0863879a5d29c750e21e94627524cc",
    "claimId": "20916bf3-8d48-475a-944e-aacb76416a71",
    "userId": "15007abe-aaf8-4687-ac94-c018e99342b5",
    "images": [{
        "8a5b76e0-5ac4-11e8-8156-2535b70a227f": {
            "vrClassification": {
                "carDamage": [{
                    "class": "car_damage_accident",
                    "score": 0.001
                }, {
                    "class": "car_damage_paint_vandalism",
                    "score": 0.001
                }, {
                    "class": "car_damage_broken_mirror",
                    "score": 0
                }, {
                    "class": "car_damage_broken_window",
                    "score": 0
                }, {
                    "class": "car_damage_flat_tire",
                    "score": 0
                }],
```

That JSON document is actually 182 lines. Example 12-2 only shows a small snippet. It is so large because the EXIF data and the weather data for the image is now being stored in Cloudant. Now, multiply that number of lines by four images that cover all sides of the car. That is a lot of data to search through for just one claim.

This situation is where creating a search index will help you find the information that you need quickly and efficiently. To do so, complete these steps:

1. Log in to IBM Cloud and click the Cloudant NoSQL DB instance that you created in an earlier module.
2. Click **Launch** to be taken to the Cloudant Dashboard.
3. Click the database name, in this example **car_accidents_3**.

The left pane has five sections: **All Documents**, **Query**, **Permissions**, **Changes**, and **Design Documents**, as shown in Figure 12-2. When indexes are added, they are created as **Design Documents**.

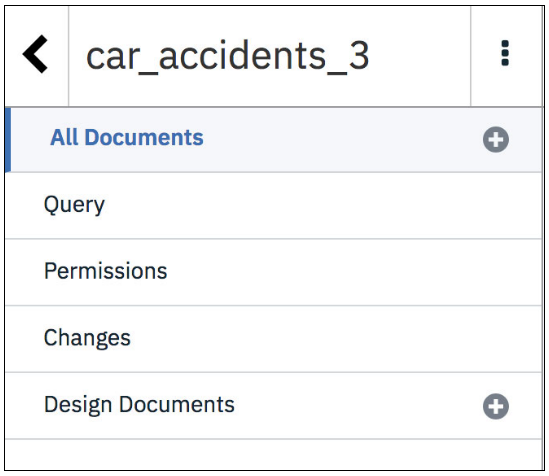
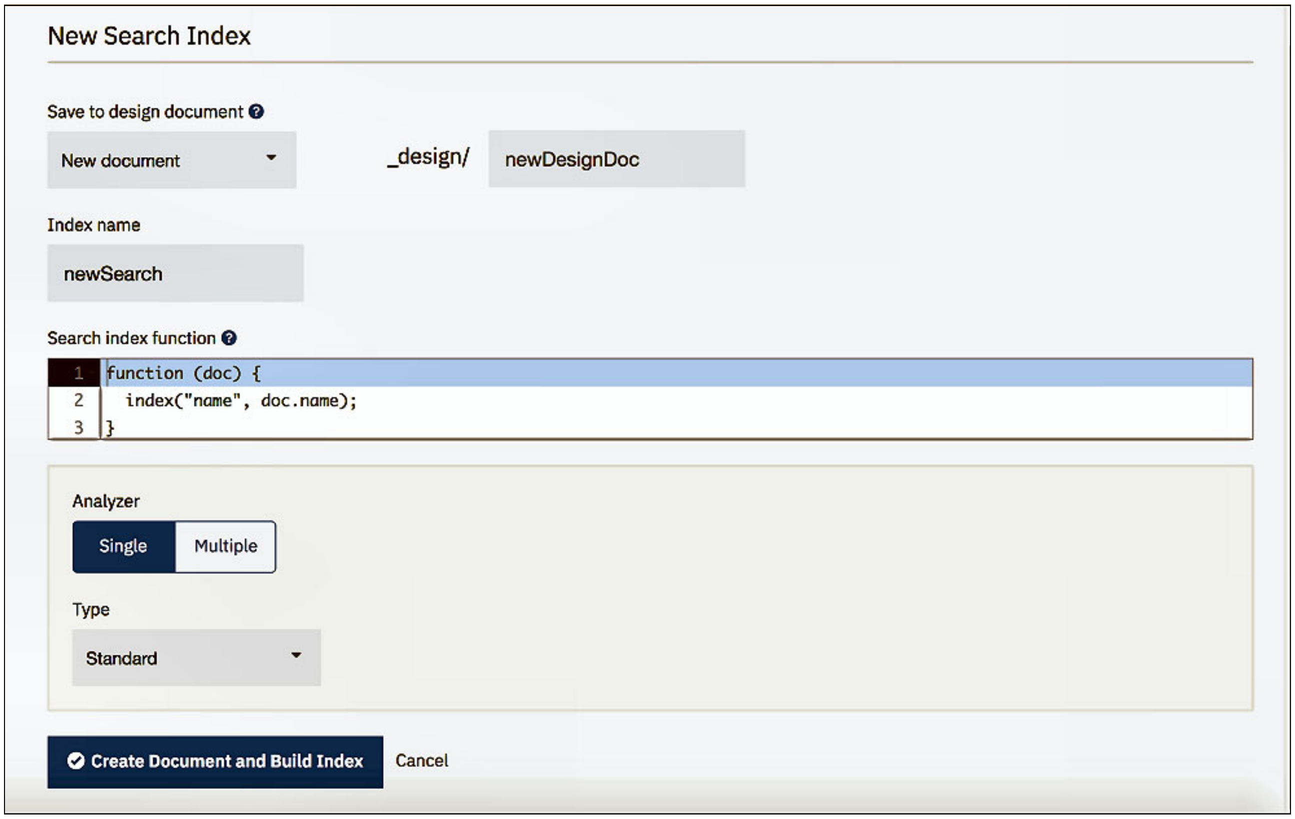


Figure 12-2 Available sections

4. Click **Design Documents +** and select **+ New Search Index**. The New Search Index window opens, as shown in Figure 12-3.



Save to design document: New document _design/ newDesignDoc

Index name: newSearch

Search index function

```
function (doc) {
  index("name", doc.name);
}
```

Type: Standard

Figure 12-3 New Search Index window

You can name the design document, so enter `module12-search` and give the index a name of `claimCreationSearch`.

We are going to create an index that allows the user to search for claims created on a specific date by using the **created** field in the Cloudant document.

5. In the **Search index function** field, enter the code in Example 12-3.

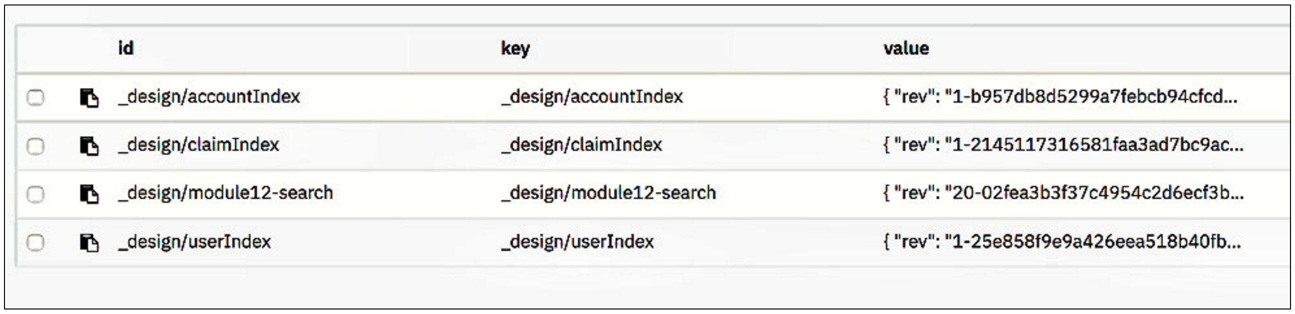
Example 12-3   Code for the Search index function field

```
function(doc) {
    if (doc.created) {
        index("created", doc.created, {
            "store": true
        });
    }
}
```

The `if` statement is called a guard clause and checks for the existence of the field in the JSON document. If it exists, it continues to build the index.

The index is on `created` and `store: true` makes sure that the value is returned in the search results. If it is false, the value will not be returned.

6. In the **Type** field, select **Keyword**.
7. Click **Save Document and Build Index**.
8. Click the Design Documents tab in the left pane. The search index that you just created will now be listed, as shown in Figure 12-4.



id	key	value
_design/accountIndex	_design/accountIndex	{ "rev": "1-b957db8d5299a7febcb94cfcd...
_design/claimIndex	_design/claimIndex	{ "rev": "1-2145117316581faa3ad7bc9ac...
_design/module12-search	_design/module12-search	{ "rev": "20-02fea3b3f37c4954c2d6ecf3b...
_design/userIndex	_design/userIndex	{ "rev": "1-25e858f9e9a426eea518b40fb...

Figure 12-4   Search index

9. Click **_design/module12-search** and view the design document for the search index. It should be similar to the document in Example 12-4.

Example 12-4   Design document example

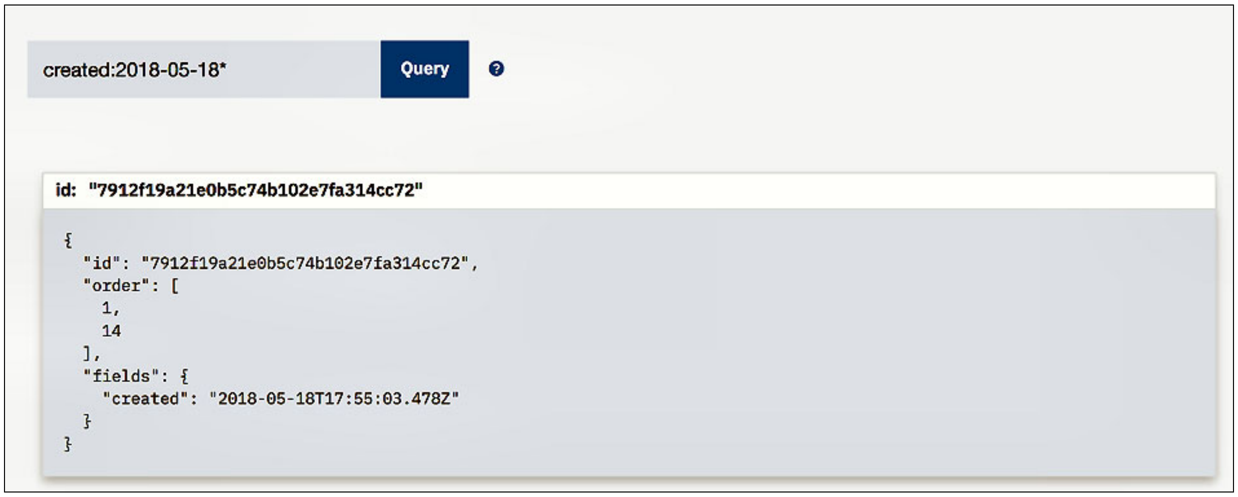
```
{
    "_id": "_design/module12-search",
    "_rev": "20-02fea3b3f37c4954c2d6ecf3b046744a",
    "views": {},
    "language": "javascript",
    "indexes": {
        "claimCreationSearch": {
            "analyzer": "keywordstandard",
            "index": "function (doc) {\n  if (doc.created) {\n index(\"created\", doc.created, {\"store\": true});\n  }\n}"
        }
    }
}
```

10. Close the search index design document and notice that there is now a **module12-search** listed in the left pane. Click **module12-search** and then click **claimCreationSearch**.

11. You will see a query field and a button labeled **Query**. Enter **created:2018-05-18*** in the field and click **Query**.

    Note: The date that you use will differ.

    The results for **2018-05-18** are displayed, as shown in Figure 12-5.



```
{
  "id": "7912f19a21e0b5c74b102e7fa314cc72",
  "order": [
    1,
    14
  ],
  "fields": {
    "created": "2018-05-18T17:55:03.478Z"
  }
}
```

Figure 12-5 Results based on date

12. Click the returned claim token to open the claim JSON document.

Queries can be added to the application's user interface to allow customers to search on specific fields and retrieve detailed information about their claims more easily.

12.4 Next steps

Review what you have learned and move on to Chapter 13, "Running your web application in the Cloud" on page 203.

12.5 Other references

- Apache Lucene

  https://lucene.apache.org/
- IBM Watson and Cloud Platform Learning Center

  https://developer.ibm.com/clouddataservices/docs/cloudant/search/
- Cloudant on YouTube

  https://www.youtube.com/user/CloudantInc/featured

13

Running your web application in the Cloud

This series has been developing an application for Insurance Company X using IBM Cloud. It has used best practices and created automation to continuously integrate and deploy your code into the cloud. This section covers how to operationalize the application.

Operating applications on the cloud requires companies to pay special attention to security, availability, reliability, and user experience. Companies also need to protect their digital assets. Ensuring that your apps are available and secure 24 hours a day, 7 days a week requires a wide array of tools and practices. This module outlines what some of these practices are and which tools you can find in the IBM Cloud to help, and also suggest some preferred practices to help you scale your apps reliably and securely.

Companies are increasingly creating teams that span the disciplines of development and operations known as DevOps. DevOps teams combine both disciplines while also practicing some form of Agile methodology and making extensive use of automation. This module does not seek to define what DevOps is. Rather, it describes some of the tools and techniques that you can use to run apps on IBM Cloud effectively.

This chapter includes the following sections:

- Learning objectives
- Getting started
- Implementation
- Next steps
- Other references

13.1 Learning objectives

At the end of this module, you should be able to:

- Describe the DevOps tools that are available in the IBM Cloud.
- Implement Internet Services to protect the application and improve scalability.
- Create an auto-scaling policy to improve the resilience of an app running in Kubernetes.
- Describe how to improve visibility of your application using logging and monitoring.

13.2 Getting started

Clone the repository to get the code:

```
$ git clone https://github.com/IBMRedbooks/IBM-Cloud-Object-Storage-Tutorials
$ cd IBM-Cloud-Object-Storage-Tutorials
```

Check out the code for Module 12 to get the finished code from the previous module that you will build on:

```
$ cd module112
```

If you want to skip ahead, check out the completed code for this module and read along:

```
$ cd module13
```

13.3 Implementation

The following sections cover the various strategies that you can use to scale and secure the web application that you built for Insurance Company X.

13.3.1 Internet Services for application security

Internet Services provides network security, reliability, and performance for internet facing applications such as the one you have built for Insurance Company X. Applications that are exposed to the public require a set of common services to enable you to deliver your application to your users:

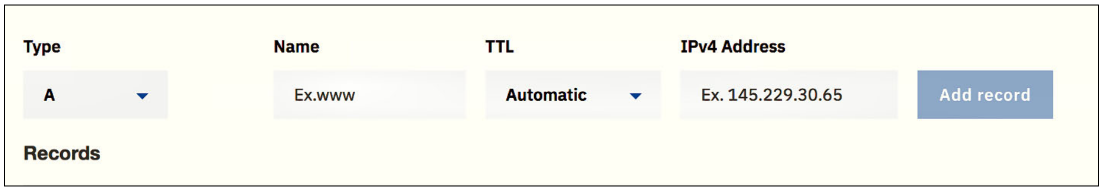
- Domain Name System (DNS): Resolve host names to a corresponding IP address. For example, the apps domain might be `insurancecompanyx.com` and you need a DNS service to translate user requests for `insurancecompanyx.com` to the IP of your apps load balancer, such as converting `insurancecompanyx.com` to `159.23.110.210`.
- SSL/TLS encryption: Encrypted data transfer between a user's device and your application.
- Caching: Provide visitors to your app with static assets from a near-by location, reducing latency and improving performance.
- Global Load Balancer: Routing traffic across servers based on their health and availability.
- DDoS protection: Protect against attempts to take down your service by flooding it with requests.
- Web Application Firewall: Protect your app against exploits and attackers.

All of the different features in this service can be orchestrated by using an API as part of your applications deployment automation.

Log in to the IBM Cloud and create an instance of Internet Services. You can find it in the catalog in the Security section.

13.3.2 Domain Name System

IBM Cloud Internet Services provide capabilities to register a domain name to the Public IP address of your load balancer in front of your application running in a Kubernetes cluster, as shown in Figure 13-1.

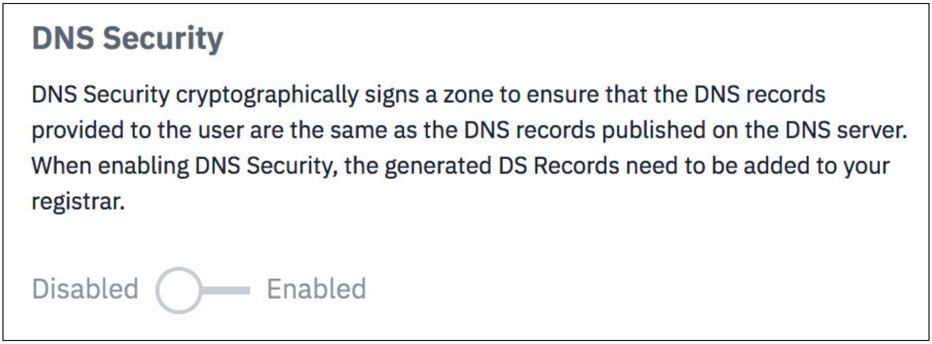


Type	Name	TTL	IPv4 Address	
A	Ex.www	Automatic	Ex. 145.229.30.65	Add record

Records

Figure 13-1 Registering domain name

Enable DNS Security, which cryptographically signs a zone to ensure that the Domain Name System (DNS) records provided to the user are the same as the DNS records published on the DNS server, as shown in Figure 13-2. This process helps to prevent DNS spoofing attacks.



DNS Security

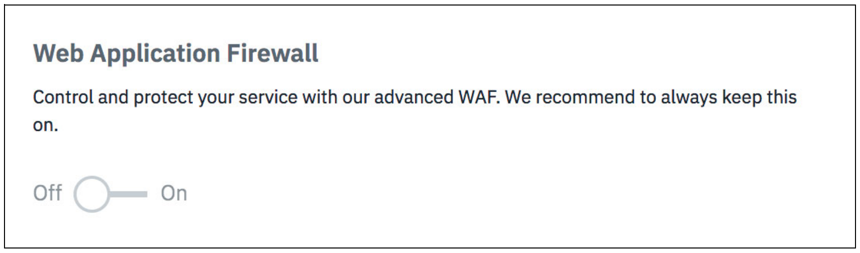
DNS Security cryptographically signs a zone to ensure that the DNS records provided to the user are the same as the DNS records published on the DNS server. When enabling DNS Security, the generated DS Records need to be added to your registrar.

Disabled Enabled

Figure 13-2 Enabling DNS Security

13.3.3 Web Application Firewall

With a click of a button, you can enable a Web Application Firewall (WAF), as shown in Figure 13-3. A WAF filters, monitors, and blocks HTTP traffic to and from a web application. A WAF can filter the traffic to and from Insurance Company X's web application. It can prevent attacks such as SQL injection, file inclusions, and other security flaws that originate from misconfigurations or mistakes in application logic. These vulnerabilities usually arise from not properly sanitizing input from a request, but they can be easily mitigated by using a WAF.



Web Application Firewall

Control and protect your service with our advanced WAF. We recommend to always keep this on.

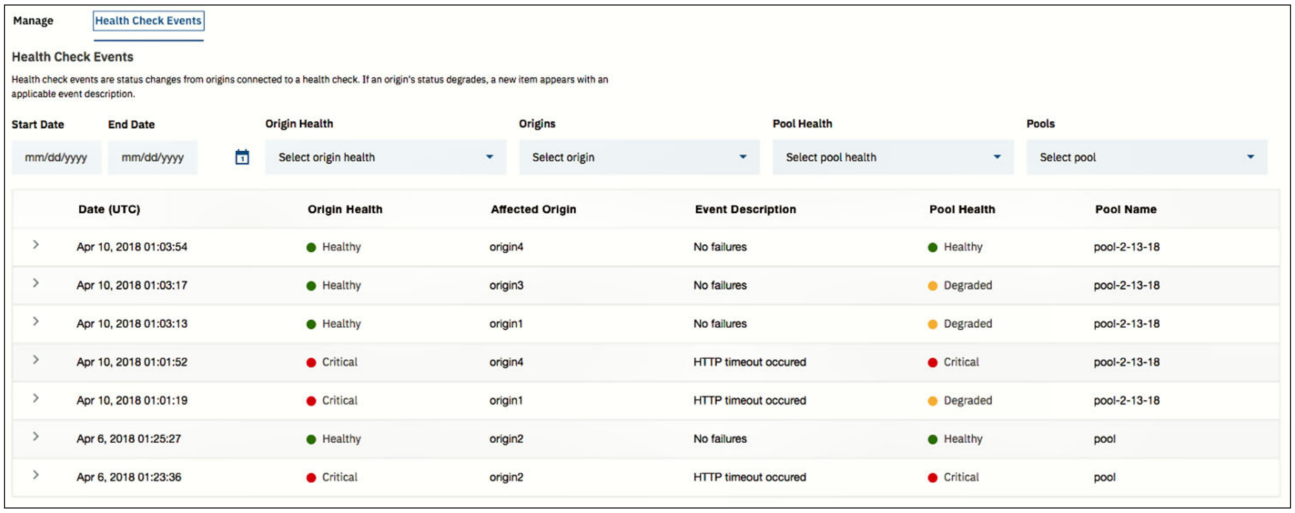
Figure 13-3 Enabling Web Application Firewall

13.3.4 Global Load Balancer

IBM Cloud is a global platform with geographically dispersed regions (EU-GB, EU-DE, US-South, and so on). To build an application with the highest possible resilience against failures, you can deploy your app into one or more regions. This will be an active → active architecture. Use a Global Load Balancer (GLB) to route traffic from users to their nearest available server. This configuration not only improves response time for the user, but also ensures that if a server or region is not available, traffic is routed to a functional region.

Applications that use this deployment topology can achieve high levels of availability. The drawback of hosting an application in this fashion is increased cost of infrastructure and additional overhead in managing that infrastructure.

Figure 13-4 shows a series of health checks that have been configured to detect the health of an application.



Health Check Events

Health check events are status changes from origins connected to a health check. If an origin's status degrades, a new item appears with an applicable event description.

Start Date	End Date	Origin Health	Origins	Pool Health	Pools
mm/dd/yyyy	mm/dd/yyyy	Select origin health	Select origin	Select pool health	Select pool

Date (UTC)	Origin Health	Affected Origin	Event Description	Pool Health	Pool Name
Apr 10, 2018 01:03:54	● Healthy	origin4	No failures	● Healthy	pool-2-13-18
Apr 10, 2018 01:03:17	● Healthy	origin3	No failures	● Degraded	pool-2-13-18
Apr 10, 2018 01:03:13	● Healthy	origin1	No failures	● Degraded	pool-2-13-18
Apr 10, 2018 01:01:52	● Critical	origin4	HTTP timeout occured	● Critical	pool-2-13-18
Apr 10, 2018 01:01:19	● Critical	origin1	HTTP timeout occured	● Degraded	pool-2-13-18
Apr 6, 2018 01:25:27	● Healthy	origin2	No failures	● Healthy	pool
Apr 6, 2018 01:23:36	● Critical	origin2	HTTP timeout occured	● Critical	pool

Figure 13-4 Sample health checks

For more information about Internet Services and different use cases you can achieve using this service, see Getting Started with IBM Cloud Internet Services (CIS).

13.3.5 Setting auto-scaling policies

You can use the horizontal pod autoscaling mechanism to increase the number of pods based on the resource utilization of the pod running on the cluster. For more information about how Kubernetes performs autoscaling, see Horizontal Pod Autoscaler.

The following command auto-scales the number of pods based on the CPU utilization of the pods. If one of the pods reaches a CPU utilization of greater than 80%, the auto scaler adds another pod to the cluster up to a maximum of five pods.

```
kubectl autoscale deployment insurancewebappbackend-deployment --min=2 --max=5
--cpu-percent=80
```

This type of policy can also be configured by using a YAML file. Open `chart/insurancewebappbackend/templates/hpa.yaml` to review how this application is configured to scale on CPU and Memory metrics.

13.3.6 Splitting the application into subdomains

Currently, the web application is built as a single monolithic application, where all the business logic is contained in one application. This configuration is rarely used in the real world. A real-world web application generally splits the business logic into separate, independently scalable entities.

Insurance Company X's web application can be restructured into different domains. For example, claims handling and user handling might be separated into different domains, and each domain placed in different pods on your cluster, as shown in Figure 13-5. These pods can be independently auto scaled by using the Kubernetes horizontal pod auto-scaler.

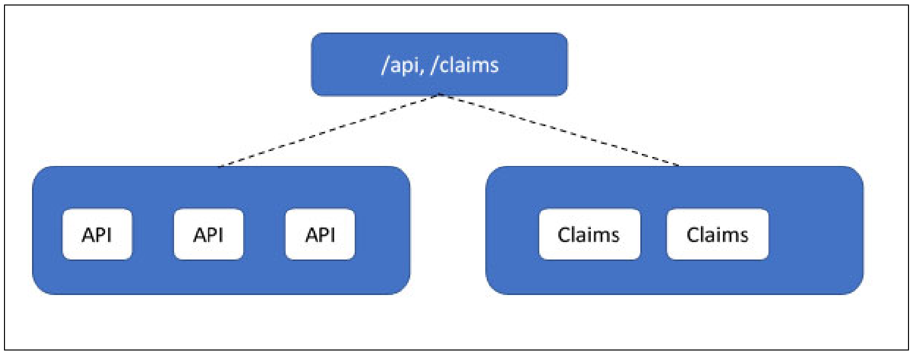


Figure 13-5 Workflow process diagram

13.3.7 Managing network traffic by using an Ingress resource

Ingress is a Kubernetes service that balances network traffic workloads in your cluster by forwarding public or private requests to your apps. You can use Ingress to expose multiple app services to the public or to a private network by using a unique public or private route.

Ingress consists of two components:

- Application load balancer
- Ingress resource

Application load balancer

The application load balancer (ALB) is an external load balancer that listens for incoming HTTP, HTTPS, TCP, or UDP service requests and forwards requests to the appropriate app pod. When you create a standard cluster, IBM Cloud Container Service automatically creates a highly available ALB for your cluster and assigns a unique public route to it. The public route is linked to a portable public IP address that is provisioned into your IBM Cloud infrastructure (SoftLayer) account during cluster creation. A default private ALB is also automatically created but is not automatically enabled.

Ingress resource

To expose an app by using Ingress, you must create a Kubernetes service for your app and register this service with the ALB by defining an Ingress resource. The Ingress resource is a Kubernetes resource that defines the rules for how to route incoming requests for an app. The Ingress resource also specifies the path to your app service, which is appended to the public route to form a unique app URL such as `mycluster.us-south.containers.mybluemix.net/myapp`.

Figure 13-6 shows how Ingress directs communication from the internet to an app.

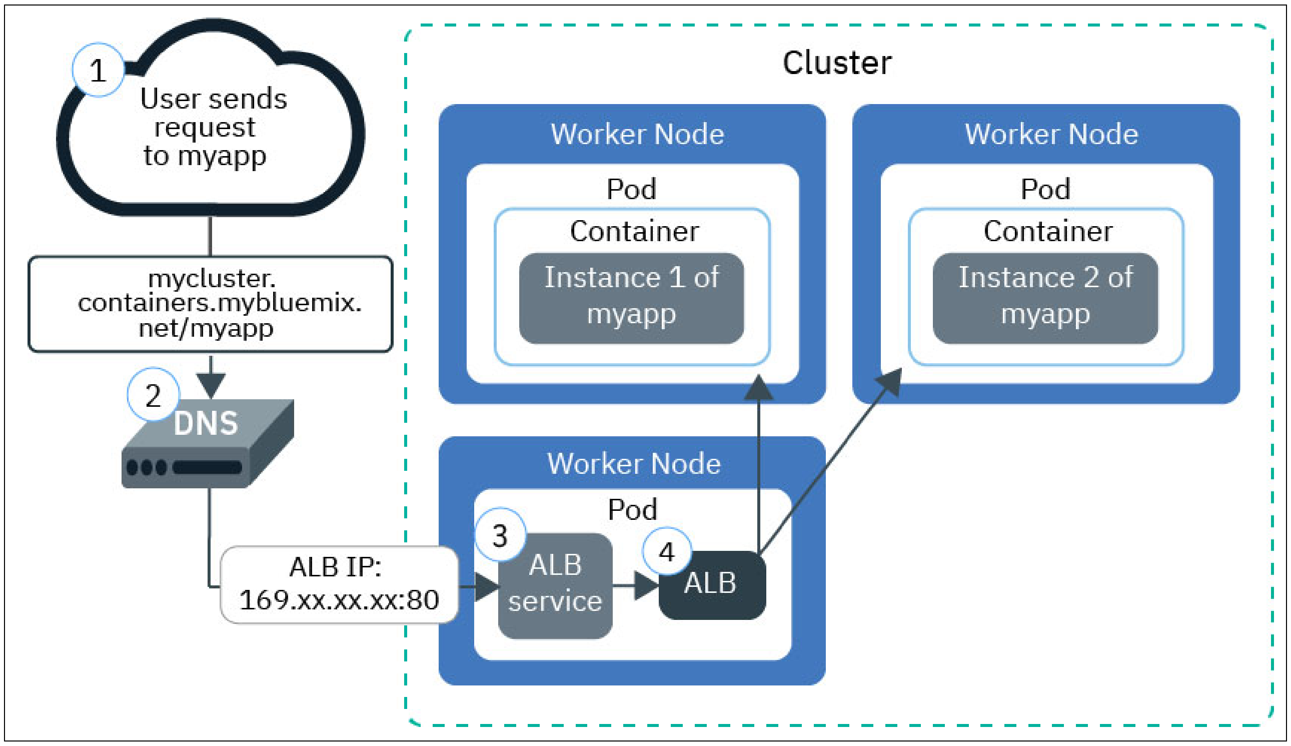


Figure 13-6 Ingress flow diagram

Prerequisites

Ingress is available for paid clusters only and requires at least two worker nodes in the cluster to ensure high availability and that periodic updates are applied. This next section will not work if you have a Lite cluster.

Setting up Ingress requires an Administrator access policy. Verify your current access policy.

An Ingress resource exposes Kubernetes services on a public or private endpoint. Example 13-1 shows the service that you are going to expose by using the Ingress.

Example 13-1 Example Kubernetes service

```
apiVersion: v1
kind: Service
metadata:
  name: insurancewebappbackend-service
spec:
  selector:
    app: insurancewebappbackend-deployment
  ports:
   - protocol: TCP
     port: 443
```

An Ingress can expose multiple apps by creating one external load balancer as a static public route, which can be an IBM provided domain or a custom domain such as `myapp.mydomain.com`. Create an Ingress resource, as shown in Example 13-2.

Example 13-2 Creating an Ingress resource

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: insurancewebappbackend-ingress
spec:
  tls:
  - hosts:
    - <cluster-name>.us-south.containers.mybluemix.net
    secretName: <cluster-name>
  rules:
  - host: <cluster-name>.us-south.containers.mybluemix.net
    http:
      paths:
      - path: /app
        backend:
          serviceName: insurancewebappbackend-service
          servicePort: 443
```

Copy and paste the code in Example 13-2 into a new file called `ingress.yaml`. Apply this Ingress resource to your cluster's namespace:

```
$ kubectl apply -f ingress.yaml --namespace=default
```

Verify that the Ingress service has been applied correctly:

```
$ kubectl get svc --namespace=default
```

You should now be able to visit the app in a browser:

```
https://<cluster-name>.us-south.containers.mybluemix.net/app
```

For more advanced configuration of the Ingress resource, such as adding your own custom domain or adding sticky sessions, see the Ingress documentation.

13.3.8 Content Delivery Network

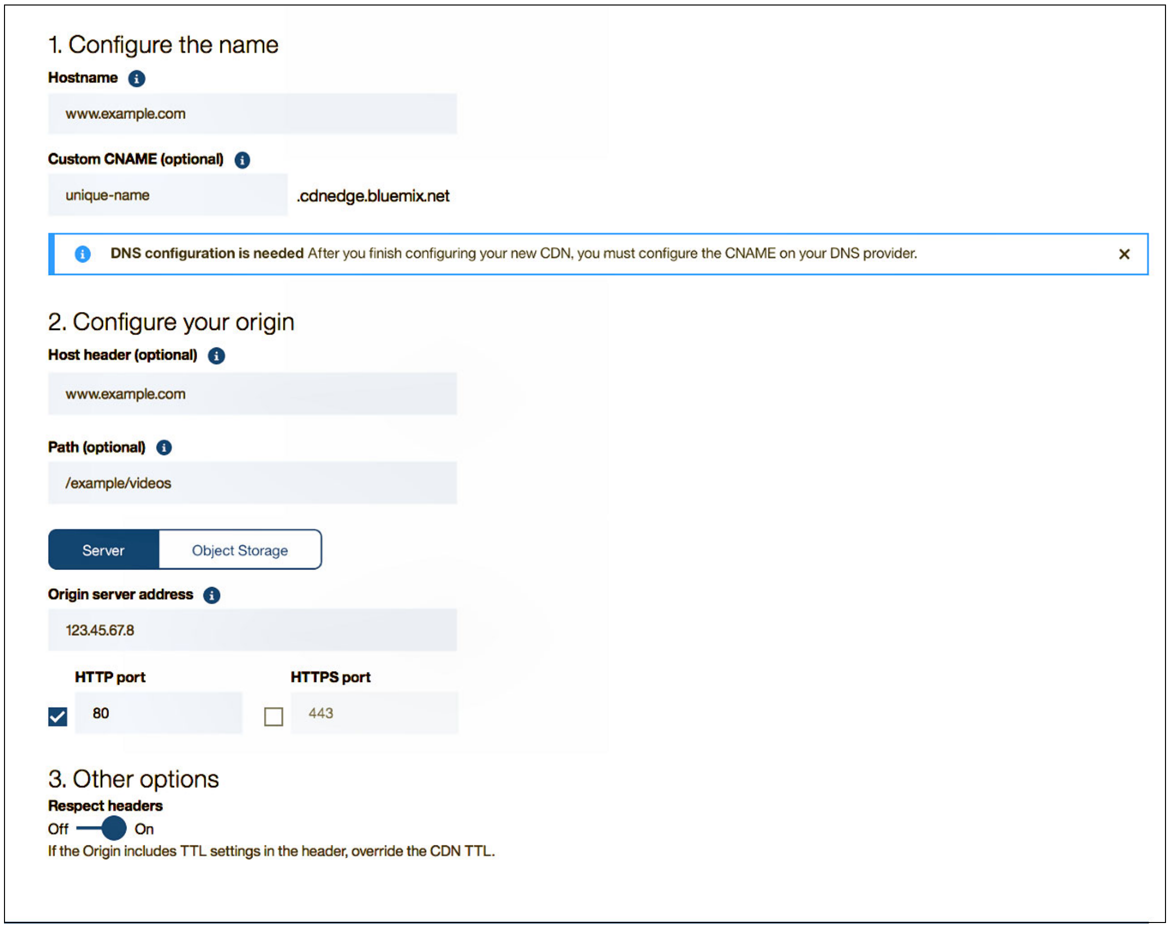
The Content Delivery Network (CDN) service of IBM Cloud distributes content where it is needed. When the content is requested for the first time, it is pulled from the host server and cached, and stays there for faster access the next time.

CDN helps you to distribute your content across geographically diverse nodes and shorten the distance that it has to travel to get to your users. This feature helps avoid network traffic jams, decrease latency, and optimize the performance of your overall cloud solution.

If Insurance Company X notices that there is a lot of traffic from a particular geographic area that is far from where the service is running, they can decide to set up a CDN service on the IBM Cloud to improve the user experience.

To configure CDN for Insurance Company X's website, complete these steps:

1. Navigate to Content Delivery Network in the IBM Cloud catalog.
2. Create an instance by entering the details shown in Figure 13-7.



1. Configure the name

Hostname: www.example.com

Custom CNAME (optional): unique-name .cdnedge.bluemix.net

DNS configuration is needed After you finish configuring your new CDN, you must configure the CNAME on your DNS provider.

2. Configure your origin

Host header (optional): www.example.com

Path (optional): /example/videos

Origin server address: 123.45.67.8

HTTP port: [✓] 80

HTTPS port: [ ] 443

3. Other options

Respect headers: Off [✓ On]

If the Origin includes TTL settings in the header, override the CDN TTL.

Figure 13-7 Creating an instance

3. Configure a CNAME with the DNS provider.

Although it might seem to be a good idea to take the images from the object storage directly with an S3 protocol, avoid doing so because the images are access controlled, which means the user images are secured in the object storage. CDN requires anonymous access to the bucket. For Insurance Company X company, we will go with the Server option and not the Object Storage option.

13.3.9 Multi Region application with Internet Services

As the Insurance Company X customer base grows and the application experiences more traffic, the developers need to start thinking about a more robust architecture. After an application becomes business critical, there will be expectations about the availability of the application. For example, the company might want a user to be able to file a claim 99.95% of the time. These increasing expectations create the challenges of high availability, disaster recovery, and ensuring a consistent user experience.

The goals are to serve the users from the region that is closest to the user, provide a service that is always available, and maintain business continuity when there is catastrophic loss to a region. You can achieve these goals by adopting a multi-region, multitier architecture:

- IBM Internet Services plays two roles in this architecture:
  - Provides data locality. Users are served from the region closest to them using intelligent routing that detects the origin of the request and selects the closest region.
  - Enhances application availability. When a region becomes unavailable, it can redirect traffic to another, functional region.
- At the application tier, no state (such as user sessions) is maintained in memory. This feature means that you can scale these components horizontally and do not need to replicate any data between regions at this tier.
- At the data tier, Cloudant provides a replication feature that enables you to maintain continuous replication between databases in different regions. However, Cloudant replication is beyond the scope of this section.
- We chose the cross-region option in Cloud Object Storage. This option means that data is dispersed across several regions, but you can use a regional endpoint to access your data. This configuration provides the high availability and disaster recovery needed to meet the requirements of a business-critical application.

Figure 13-8 shows an example disaster recovery configuration.

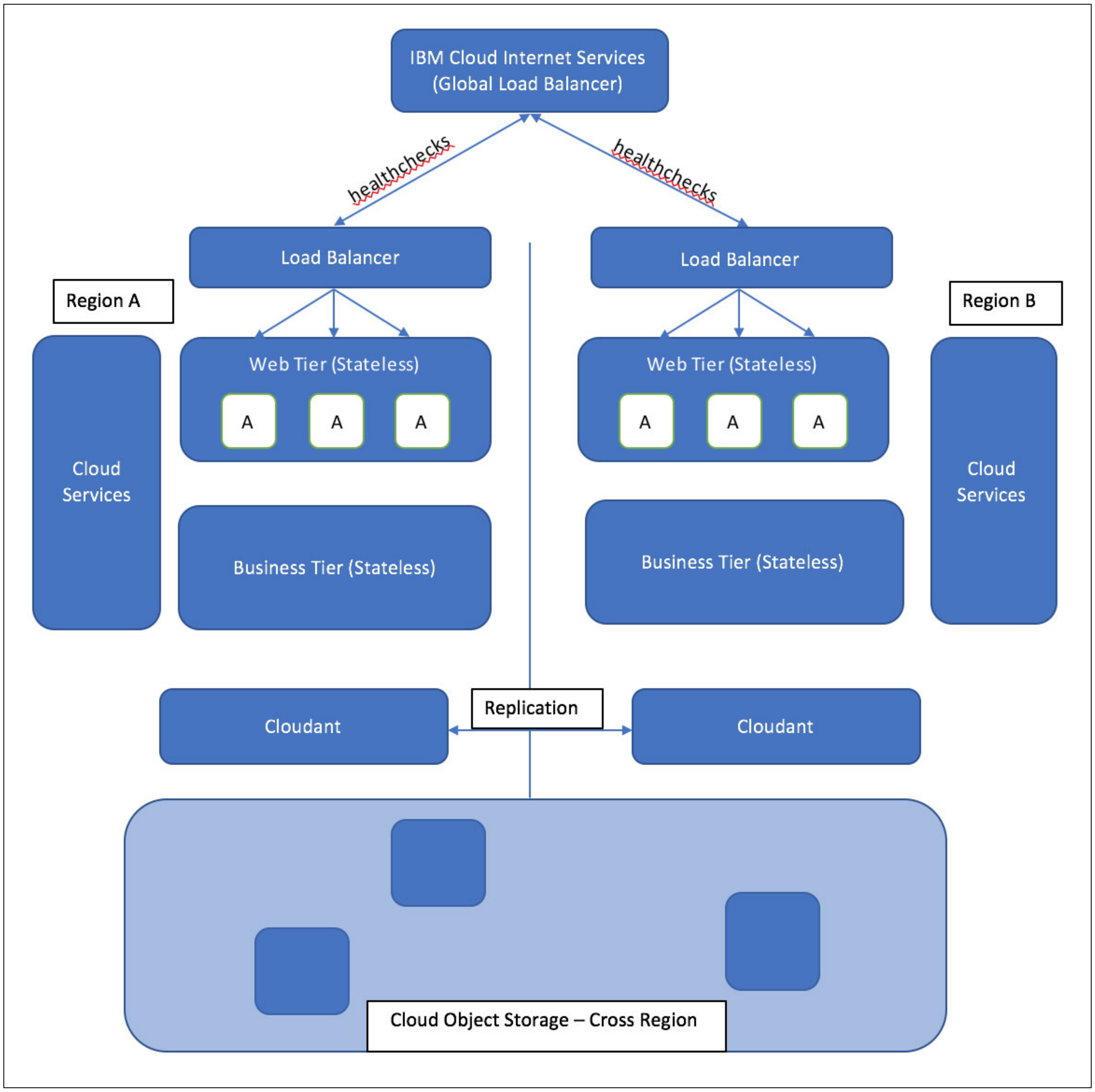


Figure 13-8 Disaster recovery configuration

13.3.10 Configuring end-to-end health checks

Health checks are a crucial part of a distributed application architecture. In our application, we have defined a `/health` API endpoint. Up until now, we consider the application healthy if, when we call the API, it is able to respond with an HTTP 200 response code, as shown in Example 13-3.

Example 13-3 HTTP 200 response code

```
var express = require("express");

module.exports = function(app) {
   var router = express.Router();

   router.get("/", function (req, res) {
      res.json({status: "UP"});
   });

   app.use("/health", router);
};
```

The code snippet in Example 13-4 is from `server/routers/health.js`. It registers a route `/health` that, when called, returns an HTTP 200 response code. Generally, include this code in every application that you build.

Example 13-4 Registering the route /health

```
livenessProbe:
  httpGet:
    path: /health
    port: {{ .Values.service.servicePort }}
  initialDelaySeconds: {{ .Values.livenessProbe.initialDelaySeconds}}
  periodSeconds: {{ .Values.livenessProbe.periodSeconds}}
```

A liveness probe defined in `chart/<project_name>/templates/deployment.yaml` defines how Kubernetes can determine whether the pod is still healthy. If it does not respond after a certain period, Kubernetes will reschedule it.

In the multi-region architecture, two levels of health checks occur. The first layer is the liveness probe that was just described. This feature provides high availability within the region for your application. If you need to be able to recover from a failure within an entire region, you must be able to perform health checks at a global level. You must also configure the Global Load Balancer from Internet Services to perform health checks at the regional level.

If you refer to the diagram in Figure 13-8 on page 212, the global load balancer is also performing periodic health checks to the load balancer in each region. This feature means that you can detect the health of an entire region and fail over to the second region if needed.

Although this health check is certainly useful, it does not tell you much about the backing services that you rely on. It shows that your application is up and healthy, but what about the database or the authentication service that you rely on to store data and allow the user to log in? If those services are down, it does not matter if the application is healthy. Therefore, check the availability of backing services in the health check endpoint. The code for this process looks as shown in Example 13-5.

Example 13-5 Checking the availability of backing services

```
var express = require("express");
const serviceManager = require("../services/service-manager");

module.exports = function(app) {
   var router = express.Router();

   router.get("/", function (req, res) {

      // check the availability of cloudant
      if(serviceManager.get("cloudant")) {
         res.json({status: "UP" });
      } else {
         res.json({ status: "DOWN" });
      }
   });

   app.use("/health", router);
};
```

Replace the code in `server/routers/health.js` with the code in Example 13-5. Then, check whether you can get the initialized Cloudant from the service manager. If so, you have a healthy application and database.

13.3.11 Improving visibility by analyzing logs and monitoring application health

You can use the IBM Cloud Log Analysis service to expand your log collection, log retention, and log search abilities in the IBM Cloud. Empower your DevOps team with features such as aggregation of application and environment logs for consolidated application or environment insights, encryption of logs, retention of log data, and quick detection and troubleshooting of issues. Use Kibana for advanced analysis tasks.

The IBM Cloud logging capabilities are integrated into the platform:

- Collection of data is automatically enabled for cloud resources. IBM Cloud, by default, collects and displays logs for your apps, app runtimes, and compute runtimes where those apps run.
- You can search up to 500 MB of logs per day.
- Logs for the last three days are stored in Log Search, a component of the Log Analysis service.

You can use the logging capabilities in the IBM Cloud to understand the behavior of the cloud platform and the resources that are running in it, as shown in Figure 13-9. No special instrumentation is required to collect the standard out and standard err logs. For example, you can use logs for these goals:

- Provide an audit trail for an application
- Detect problems in your service
- Identify vulnerabilities
- Troubleshoot your app deployments and runtime behavior
- Detect problems in the infrastructure where your app is running
- Trace your app across components in the cloud platform
- Detect patterns that you can use to pre-empt actions that could affect your service SLA

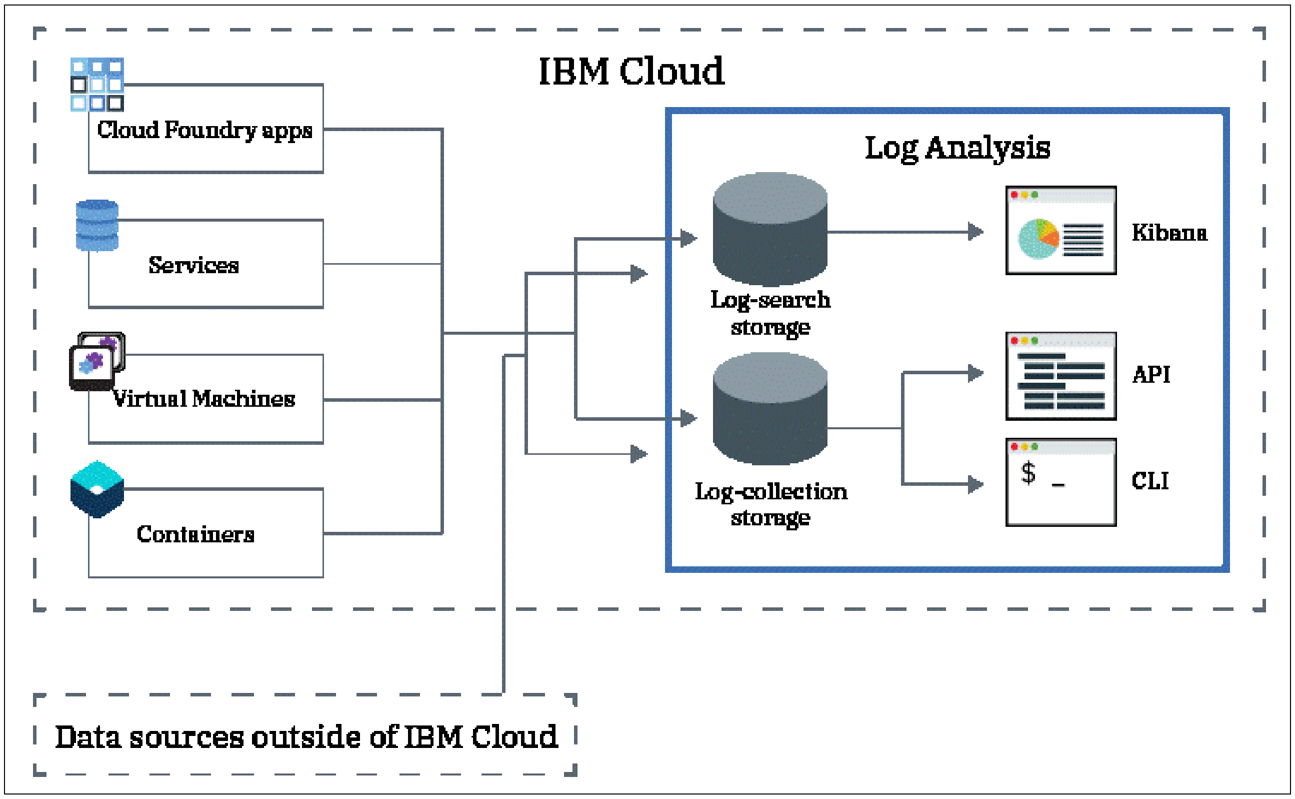


Figure 13-9 IBM Cloud logging capabilities

Follow this tutorial to set up forwarding of logs from your Kubernetes cluster and analyze them with Kibana.

13.4 Next steps

This is the last tutorial in this series. You should now be ready to create your own applications using IBM Cloud using the examples in this book.

13.5 Other references

- Horizontal Pod Autoscaler

  `https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/`

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this paper.

IBM Redbooks

The following IBM Redbooks publications provide additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *Cloud Object Storage as a Service: IBM Cloud Object Storage from Theory to Practice - For developers, IT architects and IT specialists*, SG24-8385
- *IBM Cloud Object Storage Concepts and Architecture: An Under-the-Hood Guide for IBM Cloud Object Storage*, REDP-5435
- *Building Cognitive Applications with IBM Watson Services: Volume 1 Getting Started*, SG24-8387
- *Building Cognitive Applications with IBM Watson Services: Volume 3 Visual Recognition*, SG24-8393

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Online resources

These websites are also relevant as further information sources:

- 10 Common Misconceptions about CouchDB

  https://www.youtube.com/watch?v=BKQ9kXKoHS8
- abortMultipartUpload operation

  https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#abortMultipartUpload-property
- Apache Lucene

  https://lucene.apache.org/
- Cloudant Fundamentals: The Document

  https://medium.com/ibm-watson-data-lab/cloudant-fundamentals-the-document-855c5ab92051
- Cloudant on YouTube

  https://www.youtube.com/user/CloudantInc/featured
- completeMultipartUpload operation

  https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#completeMultipartUpload-property

- createMultipartUpload operation

  https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#createMultipartUpload-property
- IBM Cloud Object Storage documentation

  https://console.bluemix.net/docs/services/cloud-object-storage/
- IBM Cloud Object Storage Node.js SDK documentation

  https://ibm.github.io/ibm-cos-sdk-js/
- IBM Cloud Object Storage Java SDK documentation

  https://ibm.github.io/ibm-cos-sdk-java
- Dms2dec

  https://www.npmjs.com/package/dms2dec
- Exiftool

  https://www.sno.phy.queensu.ca/~phil/exiftool/
- IBM Aspera high speed data transfer product information

  https://www.ibm.com/cloud/high-speed-data-transfer
- IBM Cloud Architecture Center

  https://www.ibm.com/cloud/garage/architectures/
- IBM Cloud docs

  https://console.bluemix.net/docs/
- IBM Cloud Garage Method

  https://www.ibm.com/cloud/garage/#changeHowYouWorkSection
- IBM Cloud Object Storage API Reference

  https://console.bluemix.net/docs/services/cloud-object-storage/api-reference/about-compatibility-api.html#about-the-ibm-cloud-object-storage-api
- IBM Cloud Object Storage docs

  https://console.bluemix.net/docs/services/cloud-object-storage/about-cos.html#about-ibm-cloud-object-storage
- IBM Cloud Object Storage Java SDK

  https://console.bluemix.net/docs/services/cloud-object-storage/libraries/java.html#using-java
- IBM Cloud Object Storage Node.js SDK

  https://console.bluemix.net/docs/services/cloud-object-storage/libraries/node.html#using-node-js
- IBM Cloud Object Storage Python SDK

  https://console.bluemix.net/docs/services/cloud-object-storage/libraries/python.html#using-python
- IBM Cloud Solution Tutorials

  https://console.bluemix.net/docs/tutorials/index.html#tutorials
- IBM Key Protect docs

  https://console.bluemix.net/catalog/services/key-protect
- IBM Watson and Cloud Platform Learning Center

  https://developer.ibm.com/clouddataservices/docs/cloudant/search/

- Imagemagik

  `https://www.npmjs.com/package/imagemagick`
- JSON Web Tokens versus Session Cookies

  `https://ponyfoo.com/articles/json-web-tokens-vs-session-cookies`
- Learn more about App ID

  `https://www.ibm.com/cloud/app-id`
- MIME types (a complete list)

  `https://www.sitepoint.com\mime-types-complete-list`
- More details about when to use the service

  `https://console.bluemix.net/docs/services/appid/about.html#about`
- uploadPart operation

  `https://ibm.github.io/ibm-cos-sdk-js/AWS/S3.html#uploadPart-property`
- User identity versus user account, and best practices for integrating third-party providers

  `https://cloudplatform.googleblog.com/2018/01/12-best-practices-for-user-account.html`
- The Weather Company Data Documentation

  `https://console.bluemix.net/docs/services/Weather/index.html`

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

REDP-5491-00

ISBN 0738457043

Printed in U.S.A.