Managing Microservices with Kubernetes and Istio

Intro

Poll Question

What is your experience with Kubernetes

- None
- Beginner
- Advanced
- Expert

Poll Question

What is your experience with Istio

- None
- Beginner
- Advanced
- Expert

Poll Question

What is your experience with Microservices?

- None
- Beginner
- Advanced
- Expert

Poll Question

Did you attend any of my following courses? (Choose all that apply)

- Kubernetes in 4 hours
- Kubernetes in 3 weeks
- CKAD
- CKA
- DevOps in 3 weeks
- None

Poll Question

Where are you from?

- India
- Asia (other)
- Australia / Pacific
- North / Central America
- South America
- Africa
- Europe
- Netherlands

Agenda

- Introducing Microservices and setting up Kubernetes
- Running Applications in Kubernetes
- Implementing Decoupling Using Kubernetes
- Running Microservices in Kubernetes
- Using Istio Service Mesh

Setting expectations

- This class is NOT an introduction to Kubernetes
- You don't have to be a guru, but you should at least have basic knowledge of Kubernetes
- This class teaches how to run Microservices in Kubernetes and Istio, and will explain required Kubernetes components
- For more details about Kubernetes, join my "Kubernetes in 3 weeks" or "Kubernetes in 4 hours" class
- Regarding Istio, this course provides just a short introduction to help you getting started

Related Recorded Courses

- Getting Started with Kubernetes, 2nd Edition
- Hands-on Kubernetes
- Certified Kubernetes Application Developer
- Certified Kubernetes Administrator, 2nd Edition
- **Building and Managing Microservices with Kubernetes and Istio**

Related Live Courses

- Hands on Kubernetes Deployment in 3 Weeks
- Certified Kubernetes Application Developer, 2nd Edition
- Certified Kubernetes Administrator, 2nd Edition
- Container Based Devops in 3 Weeks

Introducing Microservices

What are Microservices?

- A microservice is an application that consists of different independent components that are connected together
- The independent components can be individually developed by completely disconnected projects
- An additional layer is used that adds site-specific information to the microservices

Why use Microservices?

- Developing independent small applications is easier than maintaining one big application
- Changes can be applied faster
- By applying common DevOps technologies like Continuous Integration / Continuous Development, zero-downtime application upgrades can be provided

Microservices Benefits

- When broken down in pieces, applications are easier to build and maintain
- Each piece is independently developed, and the application itself is just the sum of all pieces
- Smaller pieces of application code are easier to understand
- Developers can work on applications independently
- If one component fails, you spin up another while the rest of the application continues to function
- Smaller components are easier to scale
- Individual components are easier to fit into continuous delivery pipelines and complex deployment scenarios

Understanding Containers and Microservices

- A container is an application that runs based on a container image. The container image is a lightweight, standalone, executable package of software that includes everything that is needed to run an application
- Because containers are lightweight and include all dependencies required to run an application, containers have become the standard for developing applications
- As containers are focusing on their specific functionality, they are perfect building blocks for building microservices
- In a containerized microservices environment, site-specific information can be provided in a flexible way by using orchestration tools such as Kubernetes

Using Kubernetes in this course

Understanding Kubernetes Delivery Options

- Kubernetes can be used as a managed service in public and private cloud
  - Amazon EKS
  - Google Cloud Platform GKE
  - Azure AKS
  - OpenStack Magnum
- Kubernetes can be used as an on-premise installation using a Kubernetes distribution
- Minikube can be implemented as a test-drive platform, mainly developed to learn Kubernetes

Understanding Kubernetes Distributions

- When using on-premise Kubernetes, something needs to be installed
- Open Source Kubernetes (AKA Vanilla Kubernetes) can easily be installed using **kubeadm** on any supported platform
- Kubernetes Distributions offer the additional value of providing support and on occasion more stable code
- Most on-premise Kubernetes distributions can be used in cloud as well

Using an environment in this course

- Any Kubernetes platform will do
- Recommended: create an Ubuntu-based minikube environment as described in the setup guide in https://github.com/sandervanvugt/microservices
- Alternatively: use O'Reilly sandbox, but functionality may be missing or different

Running applications in Kubernetes

Running Applications in Kubernetes

- Do NOT run naked pods unless it is for testing
- **kubectl create deployment** will create a deployment
  - **kubectl create deploy mynginx --image=nginx --replicas=3**
- While creating a deployment, Pods are also automatically created
- The pods are managed by the deployment: if something happens to the pod, the deployment will repair it
- Deployments also are used to support zero-downtime application updates

Understanding Decoupling in Microservices

- In microservices, it's all about independent development cycles, allowing developers to focus on their chunk of code
- To make reusing code easy, separation of static code from dynamic values is important
- This approach of decoupling items should be key in your Microservices strategy

Decoupling Resource Types in K8s

- ConfigMap: used for storing variables and config files
- Secrets: used for storing variables in a protected way
- PersistentVolumes: used to refer to external storage
- PersistentVolumeClaims: used to point to the storage type you need to use
- Service: provides access to applications running in the pods

Running Applications the Declarative Way

- To run containers, the *imperative* approach can be used
  - **kubectl create deployment**
- Alternatively, the *declarative* approach can be used, where specifications are done in a YAML file
- The declarative approach is preferred in a DevOps environment, as versions of YAML manifests can easily be maintained in Git repositories

Working with YAML Files

- Use **--dry-run=client -o yaml** with redirection to automatically generate YAML files
  - **kubectl create deploy mynginx --image=nginx --replicas=3 --dry-run=client -o\ yaml > mynginx.yaml**
- Use **kubectl explain deployment.spec** to find additional properties that can be specified in the YAML files
- Apply the YAML file
  - **kubectl apply -f mynginx.yaml** creates the resource and updates if it already exists
  - **kubectl create -f mynginx.yaml** creates the resource and fails if it already exists

Essential Troubleshooting

- **kubectl get** gives an overview of different object types and should be used as the first step in troubleshooting
- **kubectl describe** explores Kubernetes resources as created in the Kubernetes etcd database
- **kubectl logs** shows the STDOUT for applications running in a container and is useful for application troubleshooting
- **kubectl get events** gives an overview of recent events

Implementing Decoupling using Kubernetes

Understanding Pod Storage

- Volumes are a part of the Pod spec
- From a Pod, specific storage can be addressed directly
- For increased flexibility and decoupling, it is recommended to work with external storage using Persistent Volume (PV) and Persistent Volume Claim (PVC)
- PV is a Kubernetes object that connects to different types of external storage
- PVC contains a specification of required storage needs
- The Pod Volume Spec refers to the PVC to connect to a specific type of storage
- Using this approach guarantees that the developer doesn't have to know anything about storage specifics

Understanding ConfigMap

- In cloud native computing, site-specific parameters need to be provided by the cloud
- Where on host-based computing, the site-specific parameters are stored in configuration files; in cloud-native computing these parameters are stored in the cloud
- Kubernetes provides the ConfigMap resource to store site-specific information in the etcd cluster
- ConfigMap is used to provide variables and configuration files
- Secrets are encoded ConfigMaps; using secrets adds an additional layer of protection (but not encryption)
- Hashicorp Vault is a common (3rd party) solution to provide security on secrets

Procedure Overview: Using ConfigMaps

- Start by defining the ConfigMap and create it
  - Consider the different sources that can be used
  - **kubectl create cm myconf --from-file=my.conf**
  - **kubectl create cm variables --from-env-file=variables**
  - **kubectl create cm special --from-literal=VAR3=cow --from-literal=VAR4=goat**
  - Verify creation, using **kubectl describe cm <cmname>**
- Depending what is inside the ConfigMap, it will be used in the Pod specification
  - **envFrom** is used for ConfigMaps that provide environment variables
  - **volumes** are mounted for ConfigMaps that provide configuration files

Using Secrets

- Secrets are used to store sensitive information
- Kubernetes provides 3 secret types:
  - generic: general purpose, for storing passwords and other sensitive data
  - tls: for storing TLS key material
  - docker-registry: for storing Docker registry access credentials

Demo: Using ConfigMaps with Variables

- Note: it is recommended to use StatefulSet to run replicated DB
- **kubectl create deploy mydb --image=mariadb --replicas=3**
- **kubectl get pods** # it will fail
- **kubectl describe pod mydb-*nnn-mmm***
- **kubectl logs mydb-*nnn-mmm***
- **kubectl create cm mydbcm --from-literal=MARIADB_ROOT_PASSWORD=password**
- **kubectl set env --from=configmap/mydbcm deploy/mydb**
- **kubectl get all --selector app=mydb**
- **kubectl get pod mydb-*xxx-yyy* -o yaml**

Demo: Using ConfigMaps for Configfiles

- **cat nginx-custom-config.conf**
- **kubectl create cm nginx-cm --from-file nginx-custom-config.conf**
- **kubectl get cm nginx-cm -o yaml**
- **kubectl create -f nginx-cm.yaml**
- **kubectl exec -it nginx-cm -- cat /etc/nginx/conf.d/default.conf**

Understanding Services

- A service is an API resource that is used to expose a logical set of Pods
- It is also an abstraction that defines the logical set of Pods, and a policy for how to access them
- By working this way, Kubernetes service resources implement access to microservices
- The set of Pods that is targeted by a service is determined by a selector (which is a label)
- The Kubernetes cluster controller will continuously scan for Pods that match the selector and include these in the service

Understanding Services Decoupling

- Services are independent resources; they are only pointing to deployments using labels
- This allows for services to connect to multiple deployments, and if that happens, Kubernetes will automatically load balance between these deployments
  - This is commonly used in blue/green deployment and canary deployment migration scenarios
- Users connect to services on a fixed IP address / port combination, and the service load balances the connection to one of the endpoint Pods it services

Understanding Service Types

- ClusterIP: the default service type. As cluster IP is not reachable from the outside, it provides internal access only
- NodePort: allocates a port on the kubelet nodes to expose the service externally
- Loadbalancer: implements a load balancer for direct external access. No further Ingress needed
- ExternalName: a service object that works with DNS names

Understanding the ClusterIP type in Microservices

- The ClusterIP service type doesn't seem to be very useful, as it doesn't allow access to a service from the outside
- For communication of applications in a microservice architecture, it is perfect though: it does expose the application on the cluster IP network, but makes it inaccessible for external users

Understanding Service Ports

While working with services, different Ports are specified

- targetport: this is the port on the application that the service object connects to
- port: this is what maps from the cluster nodes to the targetport in the application to make the port forwarding connection
- nodeport: this is what is exposed externally

Demo: Exposing Applications Using Services

- **kubectl create deployment nginxsvc --image=nginx**
- **kubectl scale deployment nginxsvc --replicas=3**
- **kubectl expose deployment nginxsvc --port=80**
- **kubectl describe svc nginxsvc** # look for endpoints
- **kubectl get svc nginx -o=yaml**
- **kubectl get svc**
- **kubectl get endpoints**

Demo: Accessing Deployments by Services

- minikube ssh
- curl http://svc-ip-address
- exit
- kubectl edit svc nginxsvc

```
  …
  protocol: TCP
  nodePort: 32000
type: NodePort
```

- kubectl get svc
- curl http://$(minikube ip):32000

Understanding Ingress

- Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster
- Using Ingress is NOT mandatory: it can also be replaced with a simple loadbalancer
- Traffic routing is controlled by rules defined on the Ingress resource
- Ingress can be configured to do the following
  - Give services externally reachable URLs
  - Load balance traffic
  - Terminate SSL/TLS
  - Offer name-based virtual hosting
- An ingress controller is used to realize Ingress
  - On Minikube, use **minikube addons enable ingress**

Understanding Ingress Controllers

- The Ingress network controller is a network specific solution that programs an external load balancer
- Many Ingress controllers exist
  - nginx
  - haproxy
  - traefik
  - kong
  - contour
- Controllers may be installed using Operators or their own specific installation procedure

Demo: Creating Ingress

Note: this demo continues on the demo about services

- **minikube addons enable ingress**
- **kubectl get deployment**
- **kubectl get svc nginxsvc**
- **curl http://$(minikube ip):32000**
- **vim nginxsvc-ingress.yaml**
- **kubectl apply -f nginxsvc-ingress.yaml**
- **kubectl get ingress** - wait until it shows an IP address, this takes up to 3 minutes
- **sudo vim /etc/hosts**
  - **$(minikube ip) nginxsvc.info**
- **curl nginxsvc.info**

Understanding **Kustomize**

- **kustomize** is a Kubernetes feature that uses a file with the name **kustomization.yaml** to store instructions for the changes a user wants to make to a set of resources
- This is convenient for applying changes to input files that the user does not control himself, and which contents may change because of new versions appearing in Git
- So it's a way to modify existing configuration files, and thus implement changes in a declarative way
- Use **kubectl apply -k ./** in the directory with the **kustomization.yaml** and the files it refers to to apply changes
- Use **kubectl delete -k ./** in the same directory to delete all that was created by the Kustomization

Demo: Using Kustomization

- **cat deployment.yaml**
- **cat service.yaml**
- **kubectl apply -f deployment.yaml service.yaml**
- **cat kustomization.yaml**
- **kubectl apply -k .**

Lab: Microservice on Kubernetes

- Create a microservice that consists of a Wordpress application and a MySQL application, that meets the following requirements
  - Both applications are using a PVC to refer to the storage that is offered by using the default StorageClass
  - Create one yaml file for each, such that it is easy to refer to the appropriate resources from the kustomization.yaml
  - A kustomization.yaml is used to generate a secret that contains the MySQL root password as well as a configMap that specifies the name of the Wordpress host as a variable
  - The kustomization.yaml is used to refer to the WordPress as well as the MySQL resource configurations

Lab Solution

- **lesson6lab/kustomization.yaml**
- **lesson6lab/wordpress-deployment.yaml**
- **lesson6lab/mysql-deplyment.yaml**
- **kubectl get secrets**
- **kubectl get deployments**
- **kubectl get pvc** (will take few minutes)
- **kubectl get pods** (wait until all are running)
- **kubectl get services wordpress**
- **minikube service wordpress --url**
- From Minikube host: http://$(minikube ip):$(serviceport)

Running Microservices in Istio

What is a Service Mesh?

- A service Mesh can be used to manage and monitor traffic between services and microservices
- It takes case of service-to-service communication by adding additional resources and working with proxies
- Consider it a routing and tracking service: it keeps track of routing rules and dynamically direct and redirects traffic when needed

Understanding Istio Workings

- The core component in Istio is an Envoy proxy, which is injected in each Pod as a sidecar container
- These envoy proxies intercept the network traffic that enters and leaves the containers, and reroute the traffic over a dedicated service network
- Routing decisions in this Istio service network are made by the Istio control plane, which has policies that decide how the routing should take place
- The control plane is implemented as a set of Istio resources, that are added to Kubernetes using Custom Resource Definitions, and which are stored in the Kubernetes istio-system namespace
- Additional services like Prometheus and Grafana are used for metrics collection and data tracing

Understanding Common Istio Resources

- Virtual services (**VirtualService**) define how inbound requests are routed to specific hosts
- Destination Rules (**DestinationRule**) are optional resources that define policies that control traffic for a specific destination
- Gateways (**Gateway**) control the flow of traffic into and out of the service mesh
- Service entries (**ServiceEntry**) are used as an internal definition of a service that is maintained by Istio in its service registry
- The core Istio functionality is defined by Gateways that forward traffic to the virtual services

Understanding Virtual Services

- Virtual services is the Istio way to implement Canary deployments
- Virtual service resources make it easy to distribute traffic to different versions of an application, using the **subset** property for a **destination**
- The subset property can be configured to send a specific percentage of the inbound traffic to a specific version of the application
- Virtual services can also be used to direct traffic to multiple different destination services
- Virtual services can use filtering conditions that are based on header information in the inbound traffic, such as usernames

Understanding Destination Rules

- A destination rule defines a policy that is applied after the traffic has been routed
- By using **subsets**, different policies can be applied to different versions of the application
  - Subsets are based on an application version label
  - They look for applications with a matching label

Understanding Gateways

- Gateways are used to manage inbound traffic
- Gateways are implemented as Envoy proxies that run independently
  - **kubectl get pods -n istio-system**
- Virtual Services are connected to Gateways using the gateway property
  - **kubectl get virtualservice -o yaml**
- Gateways can also be used to define which traffic is allowed to leave the service mesh

Understanding Service Entries

- A service entry is an Istio internal representation of an external service
- This allows you to apply policies to outgoing traffic

Getting Started with Istio

- The Istio documentation contains a getting started guide for the current version of Istio: https://istio.io/latest/docs/setup/getting-started/
  - Consult it for details about requirements in specific Kubernetes cluster environments
- To get started with Istio, the following steps need to be performed
  - Download Istio
  - Install Istio
  - Deploy the sample application
  - Open the application to outside traffic
  - View application behaviour in the Dashboard

Understanding Istio Profiles

- Istio profiles determine the type of functionality that is installed
  - The **demo** profile is suitable for exploring Istio with minimal system requirements and high levels of tracing and access logging
  - The **default** profile is recommended for production environments
  - The **minimal** profile is setting up the control plane only
  - Specific platforms (such as OpenShift) have their own profile

Lab: Download and Install Istio

- **curl -L https://istio.io/downloadIstio | sh -**
- **cd istio-[Tab]**
- **sudo cp bin/istioctl /usr/bin/**
- **istioctl install --set profile=demo -y**
- **kubectl get all -n istio-system**
- **kubectl label namespace default istio-injection=enabled**
- **kubectl get crds**

Understanding the Bookinfo Sample Application

- The Bookinfo application is a sample application that is provided by Istio to get familiar with Istio features
- Before deploying it, make sure you have set the label **istio-injection=enabled** on the target namespace
- Deploy it using **kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml**

Lab: Deploying the Bookinfo app

- **kubectl apply -f samples/bookinfo/platform/kube/bookinfo.yaml**
- **kubectl get svc**
- **kubectl get pods**
- **kubectl exec "$(kubectl get pod -l app=ratings -o jsonpath='{.items[0].metadata.name}')" -c ratings -- curl -sS productpage:9080/productpage | grep -o "<title>.*</title>"**
- If the last command shows "Simple Bookstore App" the application is available for service (may take a minute)

Making the Application Accessible

- At this point, the application is running but not yet accessible
- To make it accessible from the outside, an Istio Gateway needs to be deployed
- The Bookinfo app contains a standard gateway that makes it accessible

Lab: Making the Application Accessible

- **less samples/bookinfo/networking/bookinfo-gateway.yaml**
- **kubectl apply -f samples/bookinfo/networking/bookinfo-gateway.yaml**
- **istioctl analyze**
- The **istioctl analyze** command must show "No validation issues found..." before continuing - this might take a minute

Setting Ingress IP and Ports

- At this point you need to set the Ingress IP address and Ports
- As the configuration is different for each type of Kubernetes cluster, you should read the section, "Determining the ingress IP and ports," in the getting started guide for specific details
  - If an external load balancer is used, that should be addressed
  - If not, you'll have to use the service NodePort
- See "Determining the ingress IP and ports" in the Istio documentation for details about setting up INGRESS_HOST and INGRESS_PORT
  - https://istio.io/latest/docs/examples/bookinfo/
- See "Determine the ingress IP and port" for defining the GATEWAY_URL variable
  - https://istio.io/latest/docs/examples/bookinfo/

Test Application Access

- This works on Minikube only
- Use **source setup_istio_vars.sh** from the course Git repository to setup access variables
- Based on the value of the $GATEWAY_URL variable, you can now access the Bookinfo app
  - **echo "http://$GATEWAY_URL/productpage"**
  - Copy the output of this command into a browser address bar to access the application

View the Dashboard

- Istio integrates with different telemetry applications to help you understand what is going on in the service mesh
- Before continuing, you should install these applications
- Next, you need to generate traffic so that something can be shown in the Dashboard

Understanding Components

- Kiali: the observability console for Istio
- Grafana: an open source multi-platform analytics and visualization web application
- Prometheus: records real-time metrics used for event monitoring and alerting
- Jaeger: software for tracing transactions between distributed versions

Lab: Using the Dashboard

- **kubectl apply -f samples/addons**
- **kubectl rollout status deployment/kiali -n istio-system**
- **istioctl dashboard kiali**
- In the left navigation menu, select Graph, and in the Namespace drop-down, select default (may have to resize screen)
- Use **for i in $(seq 1 100); do curl -s -o /dev/null "http://$GATEWAY_URL/productpage"; done** to generate some traffic and observe what is happening in the Dashboard

Using Istio Service Mesh

What's next?

- At this point, you have a basic Istio environment up and running
- Use it now, to perform different tasks
  - Managing traffic flow: allows you to shift traffic between different versions of an application
  - Fault injection: allows you to analyze how an application behaves in certain error conditions
  - Querying metrics with the Prometheus plugin
  - Visualizing metrics with the Grafana plugin
  - Using Istio to provide external service access

Demo: Implementing Request Routing

- Access **http://$GATEWAY_URL/productpage** a few times, and notice that you see different results at each attempt
- **kubectl apply -f samples/bookinfo/networking/destination-rule-all.yaml**
- **kubectl get destinationrules [-o yaml]**
- **kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml**
- **kubectl get virtualservices -o yaml** Notice that all traffic is routed to v1, using **subset: v1**
- Access **http://$GATEWAY_URL/productpage** a few times, and notice that you'll see the same results at each attempt

Demo: Implementing Request Routing using User Identity

- In the Bookinfo App, you'll use a configuration that routes all traffic initiated by user Jason to reviews:v2
- **kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-test-v2.yaml**
- **kubectl get virtualservice reviews [-o yaml]**
- Access **http://$GATEWAY_URL/productpage** and log in as user jason. Refresh the browser a few times and observe consistent behavior
- Log in as another user and access the product page. Notice that all other users are routed to the v1 review app
- Clean up the configuration, using **kubectl delete -f samples/bookinfo/networking/virtual-service-all-v1.yaml**

Demo: Implementing Traffic Shifting

- Use **kubectl apply -f samples/bookinfo/networking/virtual-service-all-v1.yaml** to route all traffic to v1 of each microservice
- Access **http://$GATEWAY_URL/productpage** to observe that no matter how often you refresh, you always get access to the same product page
- Use **kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-50-v3.yaml** to route 50% of all traffic to v3
- Review settings using **kubectl get virtualservice reviews [-o yaml]**
- Access the product page a few times to confirm
- Use **kubectl apply -f samples/bookinfo/networking/virtual-service-reviews-v3.yaml** to route all traffic to v3
- Clean up, using **kubectl delete -f samples/bookinfo/networking/virtual-service-all-v1.yaml**

Demo: Testing External Service Access

- **kubectl apply -f samples/sleep/sleep.yaml**
- **export SOURCE_POD=$(kubectl get pod -l app=sleep -o jsonpath='{.items..metadata.name}')**
- **kubectl get istiooperator installed-state -n istio-system -o jsonpath='{.spec.meshConfig.outboundTrafficPolicy.mode}'** will show the default traffic policy mode. You may see nothing, or ALLOW_ANY as the default
- Test that access is unfiltered, using **kubectl exec "$SOURCE_POD" -c sleep -- curl -sSI https://google.com | grep "HTTP/"; kubectl exec "$SOURCE_POD" -c sleep -- curl -sI https://sandervanvugt.com | grep "HTTP/"**
- You should see http return code 200, indicating successful access, or 301 which indicates that the HTTP port has moved to HTTPS permanently

Demo: Configuring Controlled External Service Access

- **istioctl install --set profile=demo --set meshConfig.outboundTrafficPolicy.mode=REGISTRY_ONLY**
- Test that access is unfiltered, using **kubectl exec "$SOURCE_POD" -c sleep -- curl -sSI https://google.com | grep "HTTP/"; kubectl exec "$SOURCE_POD" -c sleep -- curl -sI https://sandervanvugt.com | grep "HTTP/"**
- You should see http return code 35, indicating failing access
- Create the ServiceEntry: **kubectl apply -f https://github.com/sandervanvugt/microservices/httpbin.yaml**
- **kubectl exec "$SOURCE_POD" -c sleep -- curl -sS http://httpbin.org/headers**
- Observe the Envoy metadata inserted into the headers

Demo: Accessing External HTTPS Services

- **kubectl apply -f https://github.com/sandervanvugt/google.yaml**
- **kubectl exec "$SOURCE_POD" -c sleep -- curl -sSI https://www.google.com | grep "HTTP/"**
- **kubectl logs "$SOURCE_POD" -c istio-proxy | tail** should show the recent access to google.com

Understanding Needed Configuration

- To manage an application with Istio, some minimal components are needed
- In the Microservice:
  - A Microservice that uses a service to connect to one or more deployments, using a common label
  - A label on deployment and pods to identify version information
- In Istio:
  - A Gateway and a VirtualService that connect to the Microservice application
- For connectivity:
  - Proper identification of the application access details

Demo: Managing the Course Project in Istio

- **kubectl create -f istio-canary.yaml**
- **kubectl create -f littlebird-gateway.yaml**
- **source setup_istio_vars.sh**
- **kubectl apply -f istio-xxxx/samples/addons**
- **sleep 120**
- **istioctl dashboard kiali &**
- **for i in $(seq 1 100); do curl -s -o /dev/null "http://GATEWAY_URL/"; done**

Further Learning

Related Live Courses

- Containers:
  - Containers in 4 Hours
- Kubernetes
  - Kubernetes in 4 Hours
  - Kubernetes in 3 weeks
  - CKAD Crash Course
  - CKA Crash Course
  - Container Based Devops in 3 weeks

Related Recorded Courses

- Getting Started with Kubernetes, 2nd Edition
- Hands-on Kubernetes
- Certified Kubernetes Application Developer
- Certified Kubernetes Administrator, 2nd Edition
- **Building and Managing Microservices with Kubernetes and Istio**