Building Microservices with Containers

Introduction

Agenda

- Understanding Microservices
- Using Git in DevOps
- Managing Docker Overview
- The Role of Containers in Orchestration
- Understanding Kubernetes
- Creating Container Based Microservices

Poll Question 1

Which of the following topics are most interesting for you? (Choose 3)

- Understanding Microservices
- Using Git in DevOps
- Managing Docker Overview
- The Role of Containers in Orchestration
- Understanding Kubernetes
- Creating Container Based Microservices

Poll Question 2

Rate your knowledge/experience about containers

- none
- minimal
- just attended a basic course
- some working experience
- good working experience
- lots of working experience

Poll Question 3

Rate your knowledge/experience about kubernetes

- none
- minimal
- just attended a basic course
- some working experience
- good working experience
- lots of working experience

Poll Question 4

Which job title applies to you best?

- sysadmin
- devops
- developer
- security engineer
- DBA
- network engineer
- management
- other (use group chat to specify)

Poll Question 5

Which part of the world are you from?

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

Course Setup

- A running environment with Kubernetes (use cloud hosted for easy access, or minikube for local access)
- Course github repository is available at https://github.com/sandervanvugt/microservices

Building Microservices with Containers

1. Understanding Microservices

What are Microservices?

- Microservices are grains of application code that run minimal functionality that can be isolated from other grains of application code
- The different grains are loosely coupled to eachother
- The different grains are independently developed and maintained

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 included everything that is needed to run a 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
- The main requirement is that the different components are connected to eachother the right way, by providing static values instead of variables
- These static values can be provided in a very flexible way by using container orchestration platforms such as Kubernetes

How to Break up Monolitic Applications

- While breaking up monolotic applications, several changes are often applied
  - Connection parameters to database and middleware need to be changed from hard-coded to variables that can be managed in a flexible way
  - In web applications, application calls need to be changed to fixed public DNS host names
  - Security needs to be modified, to ensure that one application is allowed to access other applications. This is known as cross-origin resource sharing
- Breaking up Monolitic applications can be made easier using different solutions that are provided in Kubernetes and other orchestration platforms

What is RESTful API

- REST stands for REpresentational State Transfer and it's an architectural style for distributed systems
- REST is based on six guiding principles
  - Client-server: the client interface is separated from the server part to increase portability
  - Stateless: the server keeps no information about the state of the request, and each request must contain all the required information
  - Cacheable: the server must indicate if data in the response can be cached
  - Uniform interface: the interface to the server is well defined and uniform
  - Layered system: an architecture should be designed as multiple hierarchical layers, where a component can not see beyond its own layer
  - Code on demand: REST allows client functionaility to be extended by downloading and executing code as applets or scripts

What is RESTful API?

- The RESTful API defines resource methods to perform a desired transistion to interact with web services
- Commonly, the HTTP GET/PUT/POST/DELETE methods are used for this purpose
- Key feature is that the RESTful API should be entered without prior knowledge beyond the initial URI
- RESTful API is the standard in microservices, as it provides an easy to use, stateless and uniform way for different parts of the microservice to interact

What is CI/CD

- CI/CD stands for continuous integration / continuous delivery/deployment
- CI/CD is a method to deliver apps in a flexible and automated way
- CI/CD makes integration of new code in a solution easy
- CI/CD introduces ongoing automation and monitoring throughout the lifecycle of applications, from integration to delivery and deployment

The Role of CI/CD in Microservices

- In a microservices oriented way of working, it is important to implement changes in an easy and non-disruptive way
- CI/CD is used to guarantee this can happen. The code typically originates in a Git repository, from which it can be deployed by using automation solutions like Dockerfile or OpenShift, which on its turn are hosted in a Git repository as well
- By hosting everything in Git repositories, it's easy to manage development and version differences

What is DevOps?

- In DevOps Developers and Operators work together on implmenting new software and updates to software
- The purpose of DevOps is to reduce the time between commutting a change to a system and the change being placed in production
- DevOps is microservices oriented by nature, as more smaller projects are so much easier to manage than one monolitic project
- In DevOps, CI/CD pipelines are commonly implemented, using anything from simple GitHub repositories, up to advanced CI/CD oriented software solutions such as Jenkins and OpenShift

Understanding the goals of this course

- In this course you'll learn how to build Microservices using containers
- We'll explore the devops cycle, starting with the Git repositories and the Docker files, continuing with a strong focus on containers and their workings, and ending in the use of Microservices in the Kubernetes orchestration platform
- This course does NOT focus on the programmatic parts of microservices, so knowledge of specific programming languages is not required
- The main goal in this course is to teach students how to build a microservices based infrastructure based on orchestrated containers

Building Microservices with Containers

2. Using Git in DevOps

Using Git in a Microservices Environment

- Git can be used for version control and cooperation between different developers and teams
- Using Git makes it easy to manage many changes in an effective way
- In a Microservices way of working, this is exactly what is needed
- For that reason, Git and Microservices are a perfect match

Understanding Git

- Git is a version control system that makes collaboration easy and effective
- Git works with a repository, which can contain different development branches
- Developers and users can easily upload as well as download new files to and from the Git repository
- To do so, a Git client is needed
- Git clients are available for all operating systems
- Git servers are available online, and can be installed locally as well
- Common online services include GitHub and GitLabs

Git Client and Repository

- The Git repository is where files are uploaded to, and shared to other users
- Individual developers have a local copy of the Git repository on their computer and use the **git** client to upload and download to and from the repository
- The organization of the Git client lives in the .git directory, which contains several files to maintain the status

Understanding Git Workflow

- To offer the best possible workflow control, A Git repository consists of three trees maintained in the Git managed directory
  - The *working directory* holds the actual files
  - The *Index* acts as a staging area
  - The *HEAD* points to the last commit that was made
- The workflow starts by creating new files in the working directory
- When working with Git, the **git add** command is used to add files to the index
- To commit these files to the head, use **git commit -m "commit message"**
- Use **git add origin https://server/reponame** to connect to the remote repository
- To complete the sequence, use **git push origin master**. Replace "master" with the actual branch you want to push changes to

Creating a Git Repository

- Create the repository on your Git server
- Set your user information
  - **git config --global user.name “Your Name”**
  - **git config --global user.email “you@example.com”**
- Create a local directory that contains a README.md file. This should contain information about the current repository
- Use **git init** to generate the Git repository metadata
- Use **git add <filenames>** to add files to the staging area
- From there, use **git commit -m "commit message"** to commit the files. This will commit the files to HEAD, but not to the remote repository yet
- Use **git remote add origin https://server/reponame**
- Push local files to remote repository: **git push -u origin master**

Using Git Repositories

- Use **git clone https://gitserver/reponame** to clone the contents of a remote repository to your computer
- To update the local repository to the latest commit, use **git pull**
- Use **git push** to send local changes back to the git server (after using **git add** and **git commit** obviously)

Uploading Changed Files

- Modified files need to go through the staging process
- After changing files, use **git status** to see which files have changed
- Next, use **git add** to add these files to the staging area; use **git rm <filename>** to remove files
- Then, commit changes using **git commit -m "minor changes"**
- Synchronize, using **git push origin master**
- From any client, use **git pull** to update the current Git clone

Building Microservices with Containers

3. A Micro-introduction to Docker

Containers Defined

- A container is an application with all of its dependencies included
- Containers always start an application, they don't sit down and wait for you to tell them what to do
- To run a container, a container runtime is needed. This is the layer between the host operating system and the container itself
- Different solutions exist to run containers
  - Docker: owns more than 80% of the market and is the de facto standard
  - LXC: Linux native containers
  - CoreOS rkt AKA podman: works with Pods and natively integrates with Kubernetes

Understanding Container Types

- System containers behave like virtual machines. They go through a complete boot process and provide VM-typical services like SSH and logging
- Application containers are used to start just one application. Application containers have become the default
- Orchestration solutions like Kubernetes focus on managing application containers and may become confused while working with system images as they don't have a default application to start

Understanding a Perfect Match

- Microservices focus on developing minimal pieces of code and joining them
- Containers are focussing on developing minimal running application components
- So both have the same objective in mind
- The only thing that needs to be added, is a layer that connect containers together
- This is going to be done by the container orchestration layer

Containers and Images

- A container is a running instance of an image
- The image contains the application code, language runtime and libraries
- External libraries such as libc are typically provided by the host operating system, but in a container is included in the images
- The container image is a read-only instance of the application that just needs to be started
- While starting a container, it adds a writable layer on the top to store any changes that are made while working with the container

Building Microservices with Containers

4. The Role of Container Orchestration

Understanding Enterprise Container Requirements

- Containers are based on the Container Runtime, a solution that has nothing to connect multiple containers together in a Microservice architecture
- Containers need to be distributed, scheduled and load balanced
- Apart from that, High Availability is required
- As well as an easy solution to provide updates without downtime
- To run containers as microservices, additional platform components are required as well, such as software defined networking and software defined storage

Understanding Microservices Platform Requirements

- To understand Microservices platform requirements, you need to understand the typical Microservices application
- Microservices applications typically drill down to a database part and an accessibility part, using REST or Web to provide accessibility

From Microservice to Platform

- Based on the Microservice overview on the previous slide, the following minimal platform requirements can be defined
  - Different databases must be running as connected applications
  - Front-end services need to be added to that, and add an accessibility layer as well
  - At the user side, accessibility must be added to different types of user requests
- Add some scalability to this, as well as some high availability, and you'll have the basic platform requirements
- And optionally, add integration of a CI/CD pipeline as well to make the cycle from source code to application complete

Pearson

Understanding the Orchestration Landscape

- Kubernetes is the leading solution and the open source upstream for many other platforms
- Docker Swarm was developed by Docker Inc and offered as Docker Enterprise - now a part of Mirantis
- Red Hat OpenShift is based on the OpenShift Kubernetes Distribution (OKD) and offers Kubernetes with strong developed CI/CD

Understanding the Leading Position of Kubernetes

- Kubernetes is the leading technology that is in nearly all container orchestration platforms
- This is because of its origins, coming from Google Borg
- Google donated the Kubernetes specifications to the open source community after running Borg internally for over a decade
- As a result, a free, stable and open source project was introduced, where all vendors could base their own solution upon
- This rocked the world of container orchestration, and resulted in Kubernetes being the leading platform for orchestration

Understanding Kubernetes Delivery Options

- Kubernetes as a managed service in public and private cloud
  - Amazon EKS
  - Google Cloud Platform GKE
  - Azure AKS
  - OpenStack Magnum
- Kubernetes as an on-premise installation using a Kubernetes distribution
- Kubernetes 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 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

Common Container Distributions

- Canonical Kubernetes: offered by the makers of Ubuntu, relatively pure and based on open source, runs on premise and in cloud
- Rancher: focus on offering Kubernetes as multi-cluster deployments. Available as on-premise and in cloud
- SUSE CaaS platform: offered by SUSE as an on-premise and private cloud solution - support for public cloud is pending
- OpenShift: developed by Red Hat, available on-premise and in cloud. Adds CI/CD features to core Kubernetes functionality

Building Microservices with Containers

5. Understanding Kubernetes

Getting Started with Minikube

- Minikube is an all-in-one Kubernetes virtual machine
- Using Minikube is recommended as it gives access to all Kubernetes features, without any limitations
- In this course you'll learn how to set up Minikube on a Fedora virtual or physical machine
- For instructions on setting up Minikube on other platforms, got to: https://kubernetes.io/docs/tasks/tools/install-minikube/
- To do the hands on parts in the next lessons, it's fine to use any other Kubernetes platform as well

Installing Minikube in Fedora

- Install the most recent version of Fedora Workstation, using 40GB hard disk space, 8GB of RAM (more is recommended), and 2 CPU cores with embedded virtualization enabled
- Notice that Oracle Virtual Box does NOT support embedded virtualization for AMD processors
- Use **git clone** https://github.com/sandervanvugt/microservices to access the GitHub repository for this course
- Run the **kube-setup.sh** script from the GitHub repository and follow the instructions it provides

Running Containers in Kubernetes

- Do NOT run naked pods unless if its for testing
- **kubectl run** will run a pod
- **kubectl create deployment** will create a deployment

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
- Kubernetes helps, by adding many resource types that focus on decoupling

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

Running Containers in Pods and Deployments

- To run containers, the *imperative* approach can be used
  - **kubectl create deployment**
  - **kubectl run**
- 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

Understanding YAML

- YAML is a markup language, where parent-child relations between objects are indicated using indentation
- The objects are key: value type objects
- If a key can have multiple values, each value is indicated with a hyphen
- Use spaces for indentation, using tabs is forbidden
- Use the following in ~/.vimrc to make it easier to create YAML
  - **au! BufNewFile,BufReadPost *.{yaml,yml} set filetype=yaml foldmethod=indent**
  - **autocmd FileType yaml setlocal ts=2 sts=2 sw=2 expandtab**

Understanding the API

- The API server provides access to Kubernetes objects
- The API is also the specification of objects and their properties
- **kubectl** is the key utility to write objects to the API
- The Kubernetes API is extensible, which makes it easy to add new objects
- This is continuously happening, which is also why a new version Kubernetes is released every 3 months

Exploring the API

- **kubectl api-resources** will show the different resources, including the API group they come from
- **kubectl api-versions** will show different objects, the API they come from, as well as the current version
- **kubectl explain** allows you to browse through the different objects in the API and explore how they are constructed
  - This is a key skill for discovering how to set up the YAML code where objects are defined

Essential Troubleshooting

- **kubectl get** gives an overview of different object types and should be used as 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 events** gives an overview of recent events

Building Microservices with Containers

6. Creating Container Based Microservices

Managing Kubernetes Registries

- If no registry hostname is specified, Kubernetes fetches images from the Docker public registry
- To manage images stored in other registries, refer the complete path to the image: **my.registry.example.com/myimage**
- For using images from private registries, authentication may be required
- Perform the authentication at the container layer, for instance by making a $HOME/.docker/config.json file

Understanding **imagePullPolicy**

- By default, images are fetched from registries
- To use local images, use **imagePullPolicy: IfNotPresent** or **imagePullPolicy: Never**
- For working with local images, you'll need to make sure all cluster nodes have local pre-pulled images
- On the Kubernetes nodes, images are managed by the container layer and typically stored in an image database
- If the container layer is Docker, use **docker images ls** for a list of images
- To set up a private registry, use the procedure offered by the container layer

Understanding Pod Storage

- Volumes are a part of the Pod container spec
- For increased flexibililty and decoupling, it is recommended to work with external storage using PV and PVC
- Persistent Volume (PV) is a non-namespaced Kubernetes object that connect to different types of external storage
- Persisten Volume Claim (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

Using Dynamic Storage Provisioning

- PVs can be created manually
- Alternatively, PVs can be dynamically provisioned
- To dynamically provision a PV, a StorageClass must be referred to in the PVC
- Storage classes are using classes that are defined by the administrator
    - It's up to the admin to decide what to use to categorize: storage type, backup type, QoS, or something else
- Storage classes are also using provisioners that connect to a specific storage type
    - Each provisioner has its own parameters
- A default storage class can be used, alternatively storage classes can be defined manually

Understanding Default StorageClass

- A Default StorageClass can be used to always specify a storage type so that it doesn't have to be defined in the PVC spec anymore
- If no specific storage class is defined, the default storage class is used
- Often, the hostPath storage type is used for this purpose
- Notice that hostPath should not be used in clusters, as continuous access to storage cannot be guaranteed!
- Using the Default StorageClass provided by a cluster is simple: nothing has to be specified in the PVC definition

Demo: Using Default StorageClass

- A default storage class is provided for each cluster type
- On Minikube, it is set to hostPath
- The example YAML file default-storageclass.yaml shows how the default storageclass can be used

Understanding ConfigMaps

- The goal of the ConfigMap is to separate configuration from code
- ConfigMaps are clear-text
- They can be used in three different ways:
  - Make variables available within a Pod
  - Provide command line arguments
  - Mount them on a location where the application expects to find a configuration file
- Secrets are encoded ConfigMaps which can be used to store sensitive data
- ConfigMaps must be created before the pods that are using them

Understanding ConfigMap Sources

- ConfigMaps can be created from different sources
  - Directories: uses multiple files in a directory
  - Files: puts the contents of a file in the ConfigMap
  - Literal Values: useful to provide variables and command arguments that are to be used by a Pod

Procedure Overview: Creating 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>**
- Use **--from-file** to put the contents of a config file in the ConfigMap
- Use **--from-env-file** to define variables
- Use **--from-literal** to define variables or command line arguments

Pearson

Procedure Overview: Using ConfigMaps

- To include *variables* from a ConfigMap:

```
envFrom:
- configMapRef:
    name: ConfigMapName
```

- To include *config files* from a ConfigMap:

```
volumes:
- configMap:
    name: configMapName
    items:
    - key: my-custom.conf
      path: default.conf
```

Pearson

Demo: Creating a ConfigMap from a File

Files are provided in the GitHub repository

- Consider the contents of the file **variables**
- Create the ConfigMap: **kubectl create cm variables --from-env-file=variables**
- Verify creation: **kubectl describe cm variables**
- Create a Pod: **kubectl create -f cm-vars.yaml**
- Check that the variables are available: **kubectl logs po/cm-vars**

Demo: Creating a ConfigMap from a Literal

- **kubectl create cm cm-literal --from-literal=VAR3=goat --from-literal=VAR4=cow**
- **kubectl get cm cm-literal**

Demo: Using ConfigMaps for

- **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 /bin/bash**
  - **cat /etc/nginx/conf.d/default.conf**

Demo: Creating Secrets

- **kubectl create secret generic my-secret --from-file=ssh-privatekey=/home/student/.ssh/id_rsa --from-literal=passphrase=password**
- **kubectl describe secret my-secret**

Understanding Services

- A service is an API object 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 how to access them
- By working this way, Kubernetes service resource implement 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 not directly connected to a deployment
- The only thing they do is watch for a deployment (or other resources) based on the selector label that is specified in the service
- This allows for services to connect to multiple deployments, and if that happens, Kubernetes will automatically load balance between these deployments
- 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
- Loadbalacer: implements a load balancer for direct external access. Only available in public cloud
- 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**
- (from host): **curl http://$(minikube ip):32000**

Understanding Ingress

- Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster
- 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**

Demo: Creating Ingress - 1

Note: this demo continues on the demo in lesson 13.3

- **minikube addons enable ingress**
- **kubectl get deployment**
- **kubectl get svc nginxsvc**
- **curl http://$(minikube ip):32000**
- **vim nginxsvc-ingress.yaml**
- **kubectl apply -f nginxsvc-ingress.yaml**
- **kubectl get ingress**
- **sudo vim /etc/hosts**
  - **$(minikube ip)   nginxsvc.info**
- **curl nginxsvc.info**
- Note this doesn't work!

Demo: Creating Ingress - 2

- Previous demo doesn't work; the point of using Ingress is to have one centralized ingress into the cluster and that doesn't work with the NodePort service type
- **kubectl edit svc nginxsvc**
  - remove **nodePort: 32000**
  - change to **type: ClusterIP**
- **vim nginxsvc-ingress.yaml**
  - change to **servicePort: 80**
- **kubectl apply -f nginxsvc-ingress.yaml**
- **curl nginxsvc.info**

Understanding Kustomize

- **kustomize** is a Kubernetes feature, included since 1.14, that uses a file with the name **kustomization.yaml** to store instructions on 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

Understanding a Sample kustomization.yaml

```
resources:              # defines which resources (in YAML files) apply
- deployment.yaml
- service.yaml
namePrefix: test-    # specifies a prefix should be added to all names
namespace: testing # objects will be created in this specific namespace
commonLabels:        # labels that will be added to all objects
  environment: testing
```

Using Kustomization Overlays - 1

- Kustomization can be used to define a base configuration, as well as multiple deployment scenarios (overlays) as in dev, staging and prod for instance
- In such a configuration, the main kustomization.yaml defines the structure:

```
- base
  - deployment.yaml
  - service.yaml
  - kustomization.yaml
- overlays
  - dev
    - kustomization.yaml
  - staging
    - kustomization.yaml
  - prod
    - kustomization.yaml
```

Using Kustomization Overlays - 2

- In each of the overlays/{dev,staging.prod}/kustomization.yaml, users would reference the base configuration in the resources field, and specify changes for that specific environment:

```
resources:
- ../../base
namePrefix: dev-
namespace: development
commonLabels:
  environment: development
```

Demo: Using Kustomization.yaml

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

Poll Question 6

Which of the following topics did not get enough coverage (choose multiple)

- Understanding Microservices
- Using Git in DevOps
- Managing Docker Overview
- The Role of Containers in Orchestration
- Understanding Kubernetes
- Creating Container Based Microservices

Poll Question 7

Which of the following topics did you like most in this course (choose multiple)

- Understanding Microservices
- Using Git in DevOps
- Managing Docker Overview
- The Role of Containers in Orchestration
- Understanding Kubernetes
- Creating Container Based Microservices

Further Learning: Live Courses

- Containers in 4 Hours: Docker and Podman
- Kubernetes in 4 Hours
- Getting Started with OpenShift
- Certified Kubernetes Application Developer (CKAD) Crash Course
- Certified Kubernetes Administrator (CKA) Crash Course

Further Learning: Recorded Courses

- Hands-on Kubernetes
- Getting Started with Kubernetes LiveLessons 2nd Edition
- Red Hat OpenShift Fundamentals 3/
- Certified Kubernetes Application Developer (CKAD)
- Certified Kubernetes Administrator (CKA)