
PRODYNA
KUBERNETES TRAINING
kubernetes
TRAINING
PARTNER
kubernetes
CERTIFIED
SERVICE PROVIDER

TARGET FOR THE TRAINING

DAY 01

1. Software Containers (Docker)
   - Properties of software containers
   - Running software containers
   - Networking
   - Persistence
   - Building images
2. Kubernetes high-level architecture
3. Setup of Azure Kubernetes Service (AKS)

TARGET FOR THE TRAINING

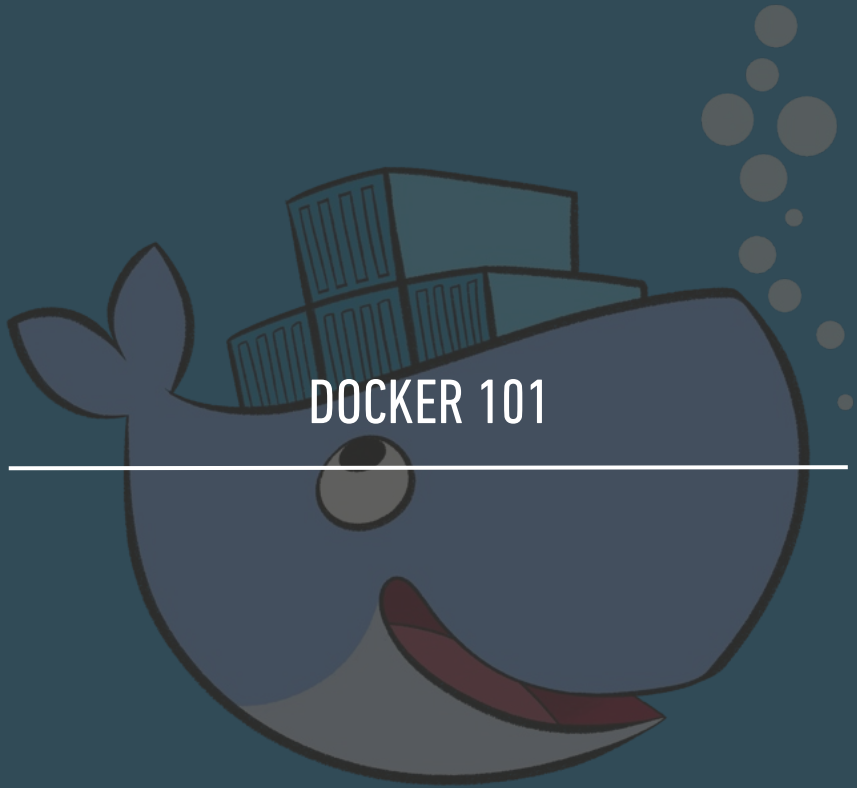
DAY 02

1. Azure Kubernetes Service (AKS) deep-dive
   - What is AKS?
   - How does it work?
   - Deployment options
2. Defining basic container workload
3. Enhanced configuration
   - Resource request / limit, Container probes, Init-containers, Scheduling
4. Controllers
   - Deployments, StatefulSets, DaemonSet
5. Network
   - Services, Ingress
   - Special network setup for Smartnet App Proxy

TARGET FOR THE TRAINING

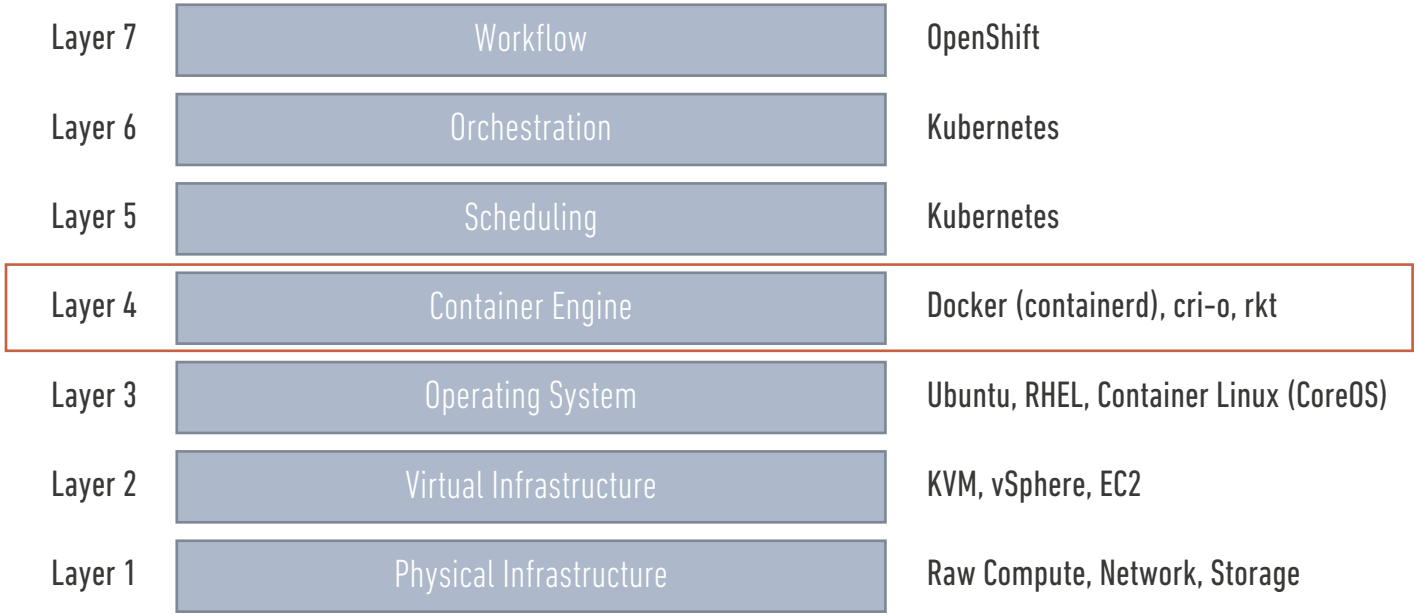
DAY 03

1. Configuration
   - ConfigMap, Secret
2. Role-based access control RBAC and Azure AD integration
3. Advanced configuration management with Helm
   - Helm chart usage
   - Helm chart creation
   - Example: Smartnet App Proxy chart
4. Smartnet App Proxy
   - Infrastructure walk-through
   - Deployment configuration
   - Observability

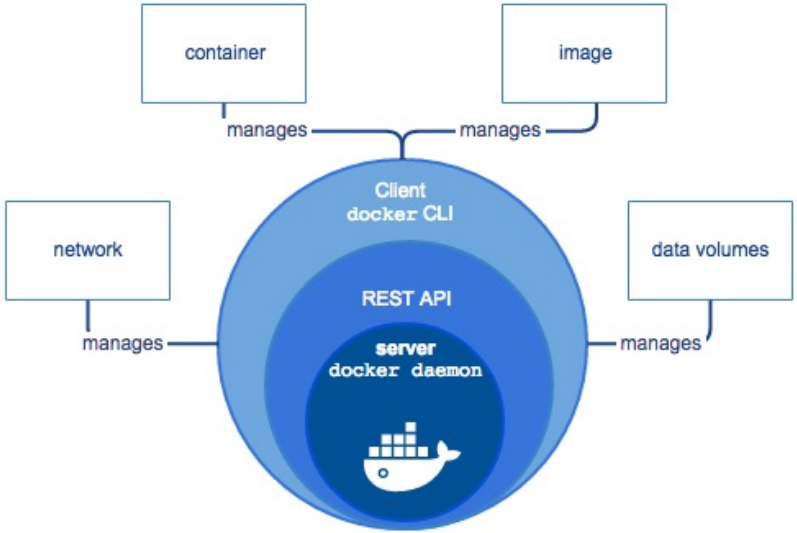


DOCKER 101

CONTAINER ECOSYSTEM LAYERS



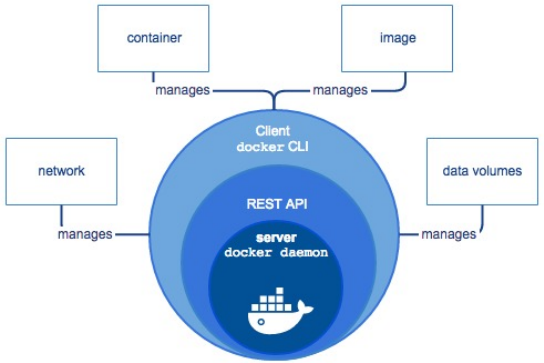
DOCKER ENGINE



Source: https://docs.docker.com

DOCKER DAEMON

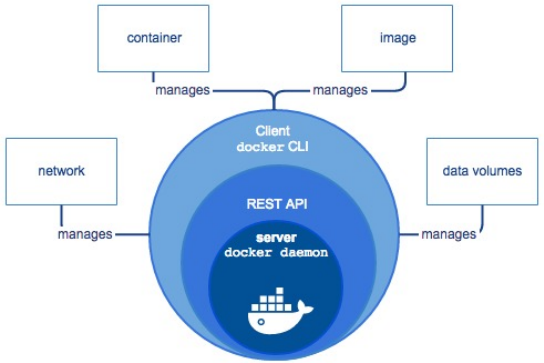
- Core component for building, managing and running containers
- Uses Linux namespaces and control group (cgroups) to isolate containers from each other and the host
- Provides dedicated network stack to each container
  - Depending on configuration containers can communicate with each other, the host and/or the rest of the world
- Requires root privileges to run
  - Rootless mode exists since Docker 19.03, but it's still alpha
- By default only reachable via Unix socket on the local machine
  - Can be configured for remote access via HTTP/REST secured by TLS



Source: https://docs.docker.com

DOCKER CLIENT

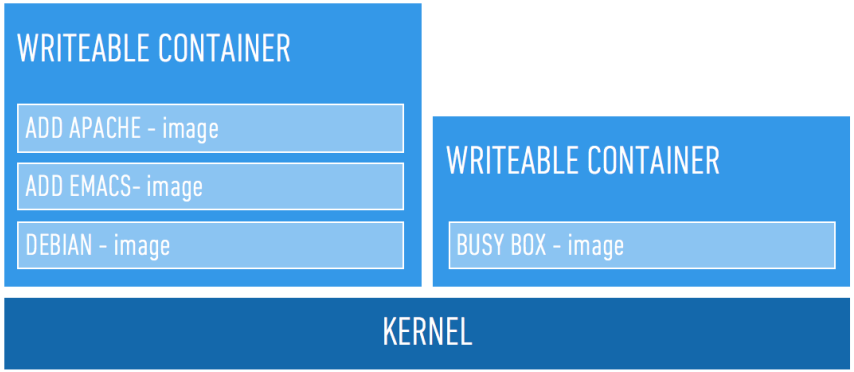
- Client for communicating with Docker daemon
- Available for most Operating Systems (Windows, macOS, Linux)
- On Linux by default uses local Unix socket to connect to Docker daemon
- Can or must use HTTP/REST interface depending on OS
- Command line reference
  - https://docs.docker.com/engine/reference/commandline/



Source: https://docs.docker.com

DOCKER IMAGE

- Docker images are the **build** component of Docker
- Read-Only template for creating containers
- Images consist of a series of layers
- UnionFS allows files and directories of separate file systems to be transparently overlaid
  → Single coherent file system
- Every image starts from a base image



DOCKER IMAGE

- Images are built from a base image using a set of instructions:
  - FROM – Definition of base image
  - MAINTAINER – Who maintains this image
  - RUN – A command to execute on top of the base image
  - ADD – Add a file or directory to the new image
  - ENV – Create an environment variable
  - ENTRYPOINT – Command to run when container is launched form this image
  - CMD – Default command and/or parameters passed to entrypoint

```
FROM java:8
ADD my-service-fat.jar app.jar
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

ALTERNATIVE TOOLS FOR BUILDING IMAGES

Google Kaniko

- Specifically created to build container images from a Dockerfile, inside a container or Kubernetes cluster
- Executes each command within a Dockerfile completely in userspace
- Multiple storage solutions for retrieving the build context are available:
  - GCS Bucket
  - S3 Bucket
  - Local Directory
  - Git Repository

Cloud Native Buildpacks

- Developed at Pivotal and Heroku for building standard based images such as OCI
- Now open-source project hosted as Incubator at CNCF
- Many advanced features (e.g. caching, multi-process, etc)
- Many integrations (e.g. Spring, GitLab, etc.)

ALTERNATIVE TOOLS FOR BUILDING IMAGES

Google Java Image Builder (JIB)

- Builds optimized Docker and OCI images for Java applications
- No Dockerfile and Docker daemon required
- Understands Java build process and separates images into different layers for optimal efficiency
  - Release dependency, Snapshot dependency, resources, classes
- Plugins available for Maven and Gradle

DOCKER CONTAINER

- Runtime instance of a Docker image
- Instantiated via Docker CLI by Docker Daemon
  - Check if image(s) are available locally
  - Download missing images from Docker Registry(s)
  - Load images into container
  - Add modifiable instance layer and mount host folders (-v option on execution)
  - Map network ports (-p option on execution)
- Can execute a single command within the container or run more complex applications as daemons

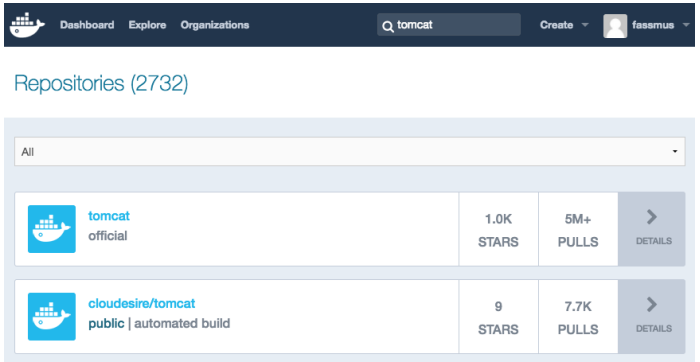
DOCKER NAMING SCHEME

- Syntax [registry][repository]:[tag]

  registry.mycompany.com/myproject/mycomponent:v1.1.0-alpine
- Image name information is used for pull and push commands to identify the repository
- Images are internally identified by hashcodes, can have multiple names/tags
- Local instances of images can be explicitly referenced with
  docker://registry.mycompany.com/myproject/mycomponent:v1.1.0-alpine

DOCKER REGISTRY

- Central component for storing and distributing Docker images
- Most current REST API definition – Docker Registry API v2
- By default, public DockerHub is used by Docker Engine
- Docker Registry Open-Source available for self-hosting
  - No authentication and authorization functionality
  - No management GUI
  - No house-keeping
- Implementations available from different vendors
  - Sonartype Nexus - https://www.sonatype.com/nexus-repository-sonatype (free or commercial)
  - Harbor - https://goharbor.io/ (open-source)
  - Artifactory - https://jfrog.com/artifactory/ (commercial)



HANDS-ON

BEST PRACTICES

EXERCISE BEST PRACTICES

- Be aware: Hands-on exercises will build upon one another
- Create your files in subfolders prefixed by slide number and name of the exercise
  - 65-pod/tomcat-sample-pod.yaml
- Adhere to the suggested names and labels
- Use only https://kubernetes.io/ as reference
- For certification preparation: Use Vim
  - (a good place to learn: https://danielmiessler.com/study/vim/)

HANDS-ON

SETUP

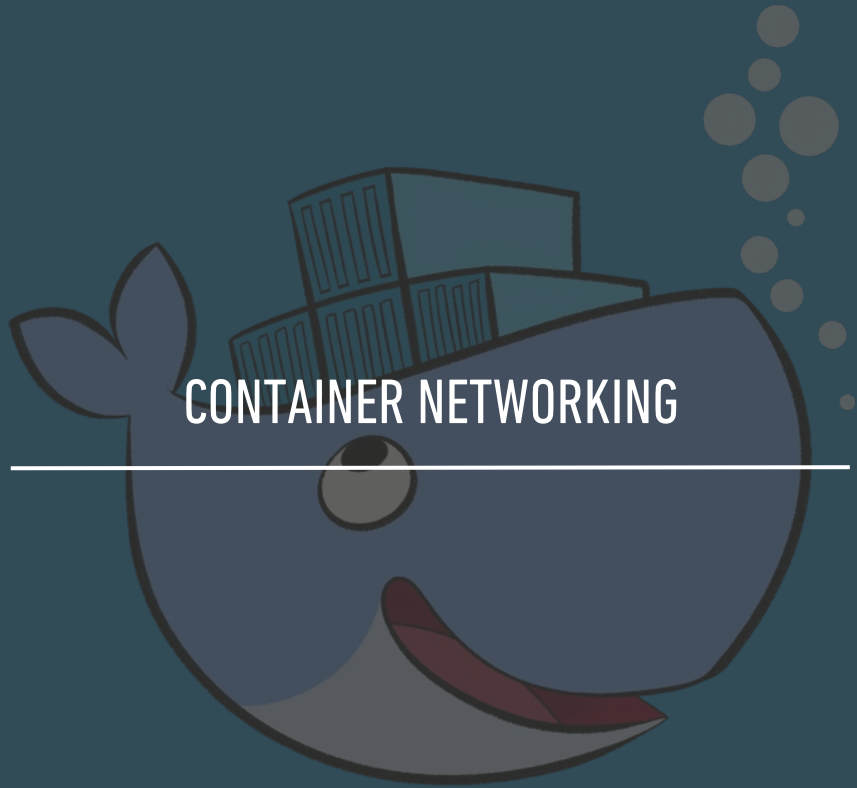
WORKPLACE

- Login to the Azure Portal
  https://portal.azure.com
  - Username:
    [ACCOUNT]@SETTEAMPRODYNA.onmicrosoft.com
  - Password:
    PRODYNA-Azure!
- Remote Desktop via Browser
  https://guacamole.dev.cnee.prodyna.com/guacamole
  - Username: [ACCOUNT]
  - Password: PRODYNA-Azure!
- Remote Desktop via RDP
  - Public IP (see list)
  - Username: ubuntu
  - Password: PRODYNA-Training!
  - Make sure to limit resolution to max. 1680×1050

Account	Participant	IP
azure-training-1	James Bridge	51.103.136.8
azure-training-2	Stefan Schneider	51.103.135.255
azure-training-3	Steven O'Neil	51.103.130.203
azure-training-4	Russell Clarke	51.103.135.188
azure-training-5	Stefan Hartmann	51.103.135.242
azure-training-6	Marco Koenig	51.103.136.7
azure-training-7	Enes Eren	51.103.133.188
azure-training-8	Michael Türtschinger	51.103.135.202
azure-training-9	Thomas Hertnagel	51.103.135.208
azure-training-10		51.103.135.244

DOCKER SETUP

3. Check version
   ```
   $ sudo docker version
   ```
4. Test installation by running hello-world
   ```
   $ sudo docker run hello-world
   ```
5. Login to PD Docker Registry
   ```
   $ sudo docker login harbor.prodyna.com
   ```
   - `Username: robot$training+push`
   - `Password: xxglSnp86XGbKdiWcZHwAg17rQ5yGXGA`



CONTAINER NETWORKING

CONTAINER NETWORKING

- Networking is provided through pluggable “Network Drivers”
  - LibNetwork
- By default Docker offers three drivers
  - bridge – Used for communication between containers on a single host
  - host – Containers will be directly connected to all interfaces available on the host
  - overlay – Network layer that spans multiple hosts and is only relevant for Swarm-Mode
- Other LibNetwork drivers for Software-Defined-Networks (SDN) available
  - Weave Net (https://github.com/weaveworks/weave)
  - Project Calico (https://github.com/projectcalico)
  - Contiv (https://contiv.io/)

CONTAINER NETWORKING

- Available default networks

```
$ docker network ls
```

```
NETWORK ID          NAME                DRIVER              SCOPE
9576121483b6        bridge              bridge              local
1d2a7029b729        dockercompose_default   bridge          local
26ff01f40f98        host                host                local
41a52974f137        none                null                local
```

- Additional networks can be defined

```
$ docker network create --driver bridge my-network
```

- Network interface can be selected on startup

```
$ docker run --network=my-network …
```

CONTAINER NETWORKING

- Ports within container can be exposed via
  `$ docker run -p 80:8080 ...`
  Expose port 8080 from container at 80 on host

  `$ docker run -p 8000-9000:8080 ...`
  Expose port 8080 from container to some port in the range 8000-9000

  `$ docker run -p 127.0.0.1:80:8080 ...`
  Expose port 8080 from container at 80 on interface 127.0.0.1 on host
- Network interfaces can be (dis)connected at runtime
  `$ docker network (dis)connect [NETWORK] [CONTAINER]`
- Networks can be inspected by
  `$ docker network inspect [NETWORK]`

HANDS-ON

CONTAINER NETWORKING

CONTAINER NETWORKING

1. Create new bridge network “my-network”

```
$ docker network create --driver bridge my-network
```

2. Inspect new network

```
$ docker network inspect my-network
```

3. Run two new Tomcat containers (detached) using above network

```
$ docker run -d -p 8080:8080 --name tomcat1 --network=my-network tomcat:8-jre8-alpine
$ docker run -d -p 8081:8080 --name tomcat2 --network=my-network tomcat:8-jre8-alpine
```

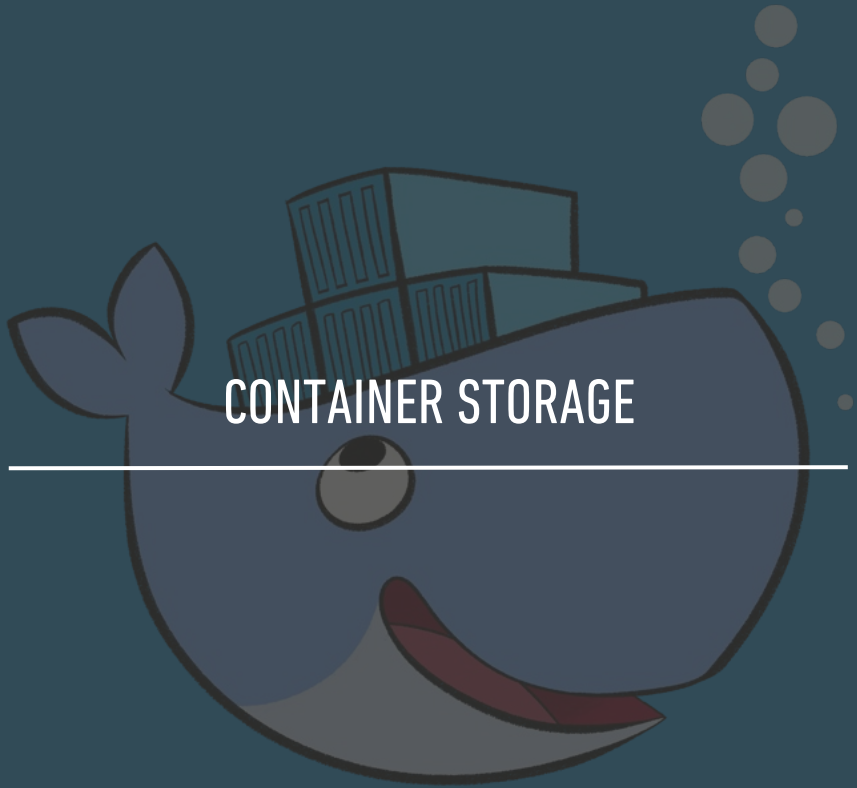
4. Check both Tomcats have started

```
$ docker ps
$ docker logs [NAME]

http://localhost:8080/
http://localhost:8081/
```

CONTAINER NETWORKING

5. Enter tomcat1 container
```
$ docker exec -it tomcat1 /bin/bash
```
6. Check network interfaces
```
$ ifconfig
```
7. Ping tomcat2
```
$ ping tomcat2
```
8. Exit tomcat1 container and disconnect tomcat2 from “my-network” network
```
$ exit
$ docker network disconnect my-network tomcat2
```
9. Repeat steps 5 – 7
10. Stop and remove both Tomcat containers
```
$ docker stop tomcat1 tomcat2
$ docker rm tomcat1 tomcat2
$ docker network rm my-network
```



CONTAINER STORAGE

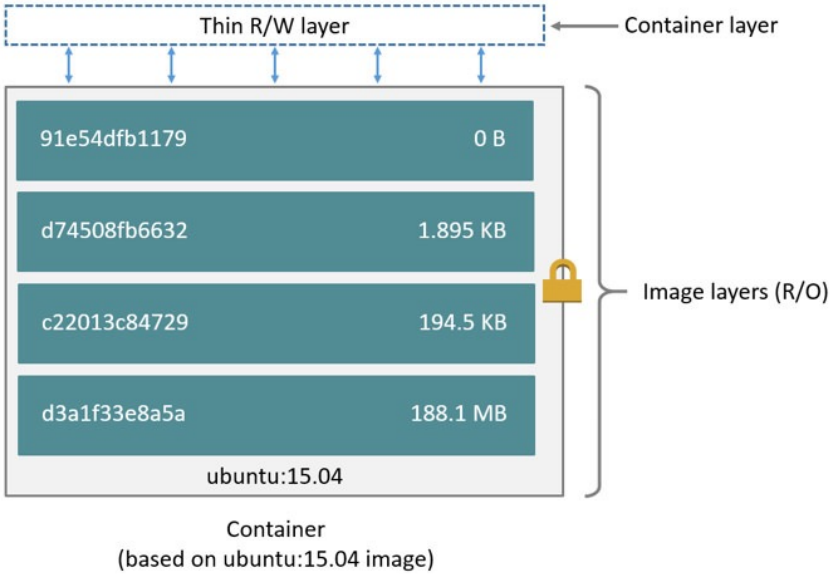
CONTAINER STORAGE

- Each Docker image references a list of read-only layers
- Each layer represent filesystem differences
- Layers are stacked on top of each other and form root FS of container

Image

CONTAINER STORAGE

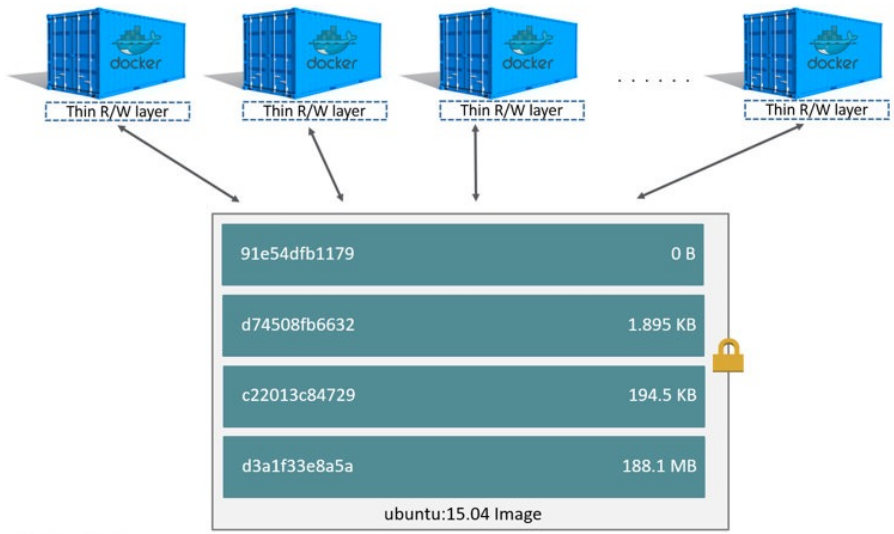
- Docker storage driver is responsible for
  - Stacking layers
  - Providing a single unified view
- When creating a new container a new, thin, writable layer on top of the underlying stack is added
- All changes made to the running container are written to this thin writable container layer
  - Writing new files
  - Modifying existing files
  - Deleting files



CONTAINER STORAGE

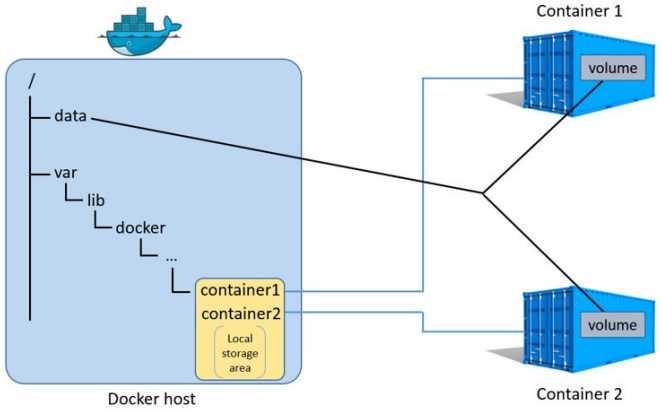
- All containers based on the same image share read-only layers
- Changes are stored in each containers writable layer
- This copy-on-write strategy enables highly efficient resource handling

- **Caution**: All data written to the writable layer will be deleted when corresponding container is deleted!



CONTAINER DATA VOLUME

- Data volumes can be used to persist data that outlives the container
- A data volume is a directory or file in the Docker host's filesystem that is mounted directly into a container
- Data volumes are not controlled by the storage driver
  - Bypass storage driver
  - Operate on host speed
- Volumes are initialized when a container is created
  - If the container's base image contains data at the specified mount point, that existing data is copied into the new volume upon volume initialization.



CONTAINER DATA VOLUME

- Adding a data volume
  `$ docker run ... -v /test ...`
  Will create a new volume and mount it as /test into the container

  `$ docker run ... -v /home/user/my-folder:/test ...`
  Mount host folder /home/user/my-folder into container at /test
- Volumes create by one container can be shared with others
  `$ docker run ... --volumes-from [CONTAINER]`
- Docker volume plugins allow provisioning and mounting shared storage
  - iSCSI
  - NFS
  - FC

On Mac and Windows only folders below /Users (macOS) or C:\Users can be mounted by default.

HANDS-ON

CONTAINER STORAGE

CONTAINER STORAGE

1. Run two new Tomcat containers as daemons (tomcat:8-jre8-alpine) and mount host folder

```
$ docker run -d --name tomcat1 -v ~/temp:/data tomcat:8-jre8-alpine
$ docker run -d --name tomcat2 -v ~/temp:/data tomcat:8-jre8-alpine
```

2. Inspect tomcat container

```
$ docker ps
$ docker inspect tomcat1
```

3. Create a new file within the share volume via tomcat1

```
$ docker exec tomcat1 /bin/sh -c 'echo "Test" > /data/test.txt'
```

4. Check for file via tomcat2

```
$ docker exec tomcat2 cat /data/test.txt
```

5. Clean up

```
$ sudo docker stop $(sudo docker ps -q)
$ sudo docker container prune
```

HANDS-ON

CONTAINER BUILD

BUILD CUSTOM CONTAINER IMAGE

1. Inspect the tomcat-docker-sample which can be found at
   ```
   $ cd ~/kubernetes-training/tomcat-sample
   ```
2. Build application
   ```
   $ mvn clean package
   ```
3. Build image
   ```
   $ docker build -t harbor.prodyna.com/training/tomcat-sample:[id] ./target
   ```
   1. id: azure-training-[number]
4. Check available images
   ```
   $ docker images
   ```
5. Run container (http://localhost:8080/tomcat-sample/)
   ```
   $ docker run -d -p 8080:8080 --name tomcat-sample harbor.prodyna.com/training/tomcat-sample:[id]
   ```

BUILD CUSTOM CONTAINER IMAGE

6. Push image to central PRODYNA Docker Registry
```
$ docker push harbor.prodyna.com/training/tomcat-sample:[id]
```
7. Stop and remove all traces of container and image locally
```
$ docker stop tomcat-sample
$ docker container prune
$ docker image prune -a
```
8. Check cleanup
```
$ docker ps -a
$ docker images
```
9. Rerun container
```
$ docker run -d -p 8080:8080 --name tomcat-sample harbor.prodyna.com/training/tomcat-sample:[id]
```
10. Clean up
```
$ sudo docker stop $(sudo docker ps -q)
$ sudo docker container prune
```

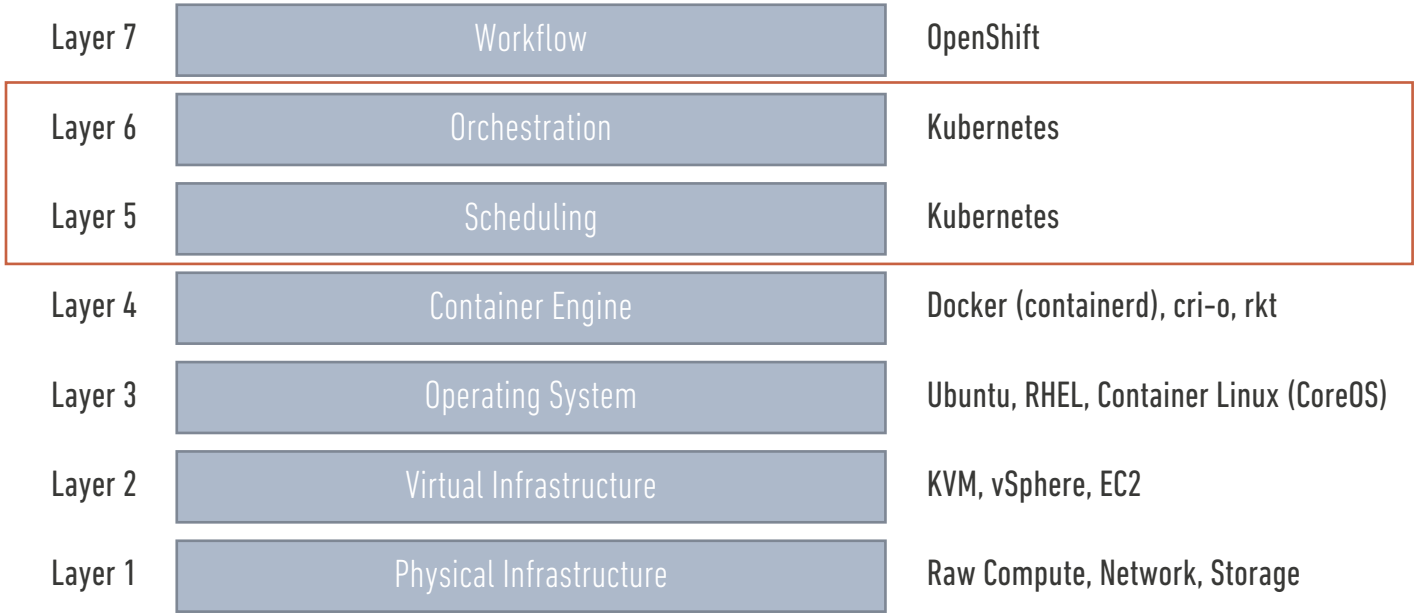
KUBERNETES

OVERVIEW

kubernetes

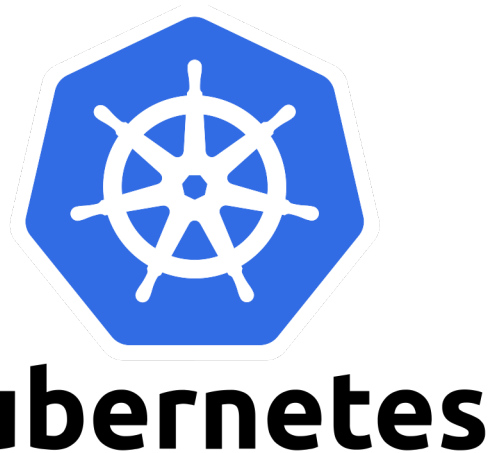
by Google

CONTAINER ECOSYSTEM LAYERS



KUBERNETES OVERVIEW

- Open-Source platform for managing containerized applications (https://kubernetes.io/)
- Builds upon 15 years of experience of running production workloads at Google
- Provides mechanisms for application
  - Deployment
  - Scheduling
  - Updating
  - Maintenance
  - Scaling
- Lean, Portable and Extensible

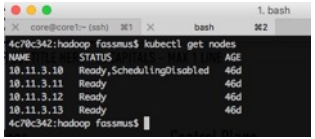


KUBERNETES UX

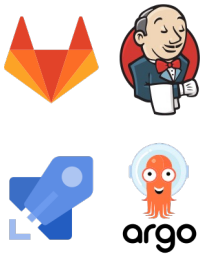
Spec

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

User



CI/CD



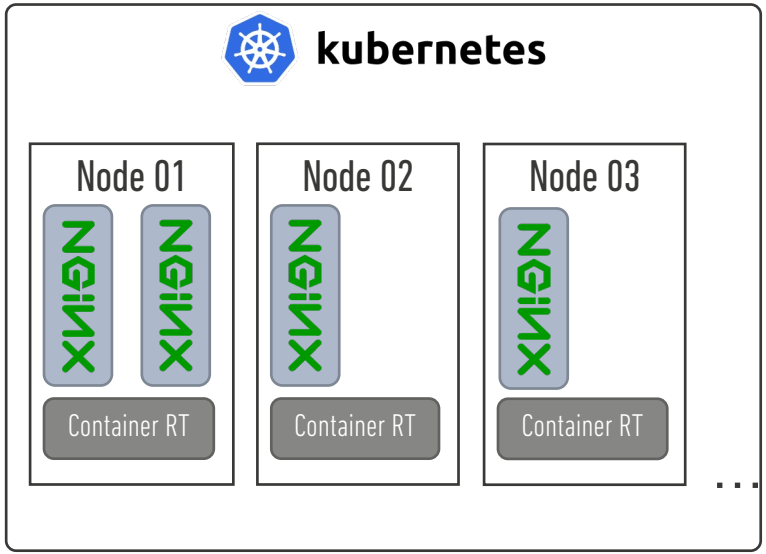
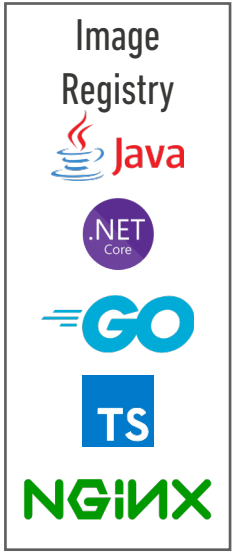
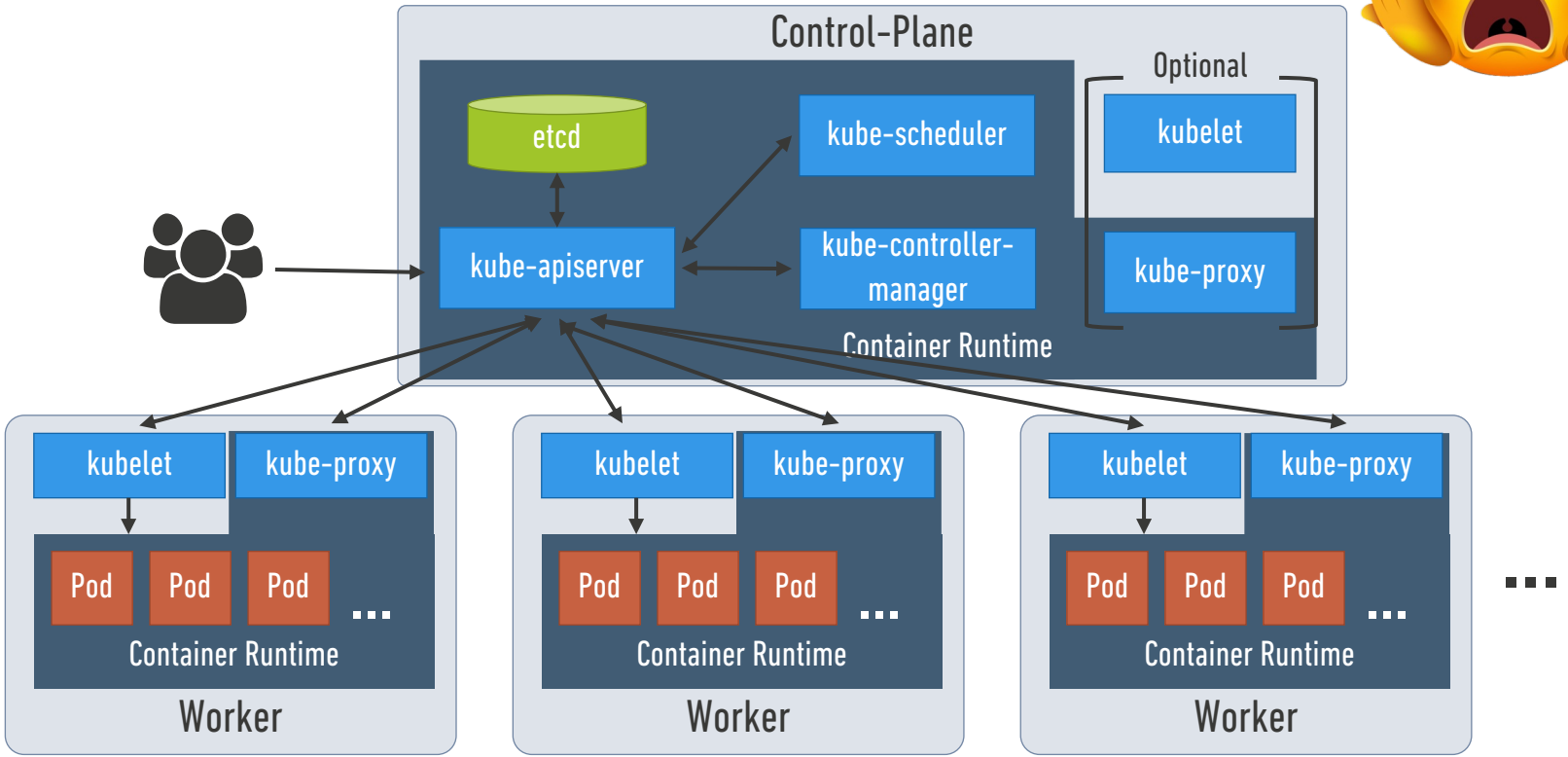


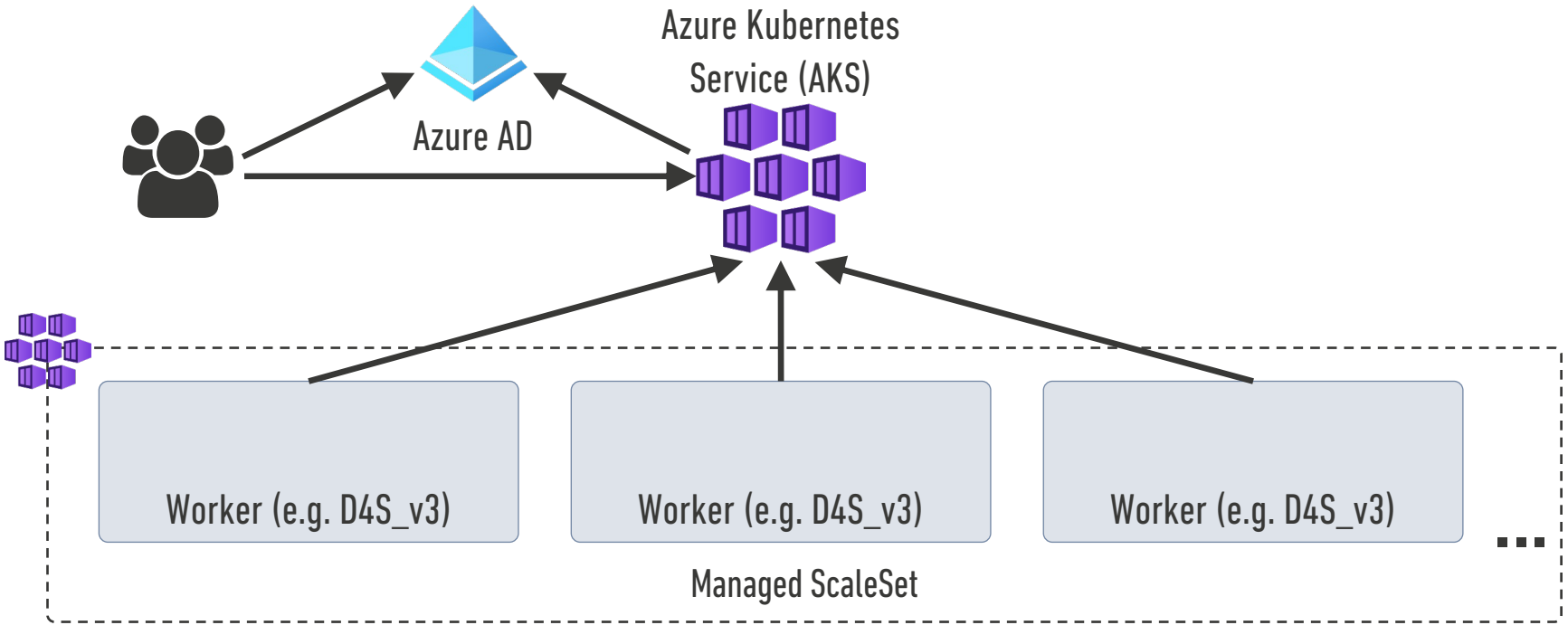
Image Registry



KUBERNETES ARCHITECTURE



AZURE KUBERNETES SERVICE



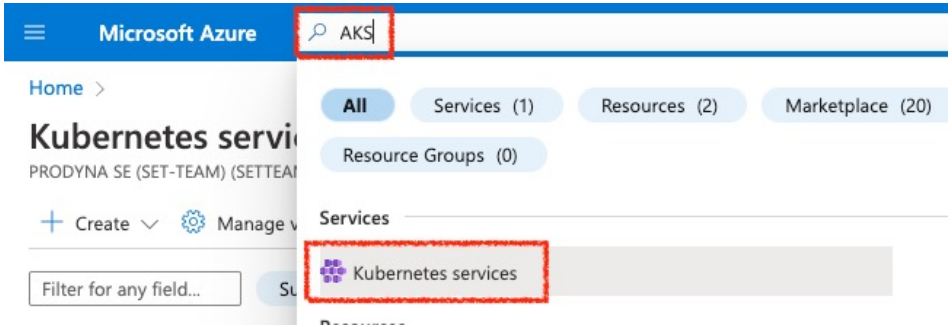
HANDS-ON

AKS SETUP

AKS SETUP

TASK - SETUP A AZURE KUBERNETES CLUSTER

1. Go to the Azure Portal and search for AKS



2. AKS general settings
   - RG: rg-training-?
   - Cluster preset Configuration: Dev/Test ($)
   - Name: aks-training-?
   - Region: West Europe
   - Leave rest at default values

AKS SETUP

TASK - SETUP A AZURE KUBERNETES CLUSTER

3. Click “Next Node pools“ and change „agentpool“
   - Name: system
   - Scaling method: manual
   - Node count: 1
4. Add additional node pool
   - Name: app
   - Node size: B2ms
   - Scaling method: manual
   - Node count: 1
5. Click “Next Access” and perform following settings:
   - Authentication and Authorization: Azure AD authentication with Kubernetes RBAC
   - Choose AAD group: Training_Users

AKS SETUP

TASK - SETUP A AZURE KUBERNETES CLUSTER

6. Click “Next Networking“ and perform following settings:
   - Network configuration: Azure CNI
   - Network Policies: Azure
   - Leave rest at default values
7. Click "Review & Create" followed by a final "Create"
8. Go to your command line and login to Azure via Azure CLI
   ```
   $ az login
   ```
9. Login to AKS to use kubectl
   ```
   $ az aks get-credentials --resource-group rg-training-? --name aks-training-?
   ```
10. Run k9s and explore cluster
    ```
    $ k9s
    ```

AKS SETUP

TASK - SETUP A AZURE KUBERNETES CLUSTER

11. Create docker-registry secret for private registry (password see ~/kubernetes-training/harbor.txt)

```
$ kubectl create secret docker-registry harbor --docker-server=harbor.prodyna.com --docker-username='robot$training+push' --docker-password='xxglSnp86XGbKdiWcZHwAg17rQ5yGXGA' --docker-email=foo@bar.com
```

12. Add secret to default service account

```
$ kubectl patch serviceaccount default -p '{"imagePullSecrets": [{"name": "harbor"}]}'
```

13. Inspect secret and service-account

```
$ kubectl describe secret harbor
$ kubectl get secret harbor -o yaml
$ kubectl describe sa default
```

AKS SETUP

TASK - SETUP A AZURE KUBERNETES CLUSTER

14. Run Tomcat sample and see what happens on k9s

```
$ kubectl run tomcat-sample --image
harbor.prodyna.com/training/tomcat-sample:[id] --port 8080
```

15. Inspect logs

```
$ kubectl logs tomcat-sample
```

16. Establish port forward to Tomcat pod

```
$ kubectl port-forward tomcat-sample 8080:8080
```

17. Call Tomcat (http://localhost:8080/tomcat-sample/)

18. Cleanup

```
$ kubectl delete pod tomcat-sample
```

KUBERNETES

CORE CONCEPTS

kubernetes

by Google

KUBERNETES CORE CONCEPTS

- Objects
- API
- Namespaces
- Labels + Label Selector
- Annotations

KUBERNETES OBJECTS

- Kubernetes Objects are persistent entities in the Kubernetes system
- Used to represent the state of the cluster
  - What containerized applications are running (and on which nodes)
  - The resources available to those applications
  - The policies around how those applications behave, such as restart policies, upgrades, and fault-tolerance
- Represents a "record of intent"
  → Desired state
- All objects have common structure
  - `apiVersion`
  - `kind`
  - `metadata`
- `spec` must be provided but differs between objects (Kubernetes API Reference)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kube-proxy-22czg
  ...
spec:
  containers:
  - name: kube-proxy
    image: k8s.gcr.io/hyperkube-amd64:v1.11.5
    ...
status:
  conditions:
  ...
  containerStatuses:
  ...
  hostIP: 10.240.0.4
  phase: Running
  podIP: 10.240.0.4
  startTime: 2018-12-17T10:20:41Z
```

KUBERNETES OBJECTS

- Pods
- Controllers
  - Replica Sets
  - Deployments
  - Stateful Sets
  - Daemon Sets
  - Garbage Collection
  - Jobs - Run to Completion
  - Cron Jobs
- Storage
  - Volumes
  - Persistent Volumes
  - Dynamic Provisioning
- Configuration
  - ConfigMap
  - Secrets
- Security
  - (Cluster)Roles
  - (Cluster)RoleBindings
  - Network Policies
- Services
  - Load Balancing, and Networking
  - Services
  - DNS Pods and Services
  - Connecting Applications with Services
  - Ingress Resources

https://kubernetes.io/docs/concepts/

KUBERNETES API

- The Kubernetes API serves as the foundation for the declarative configuration schema
- Complete API details are documented using Swagger v1.2 and OpenAPI
- Kubernetes supports multiple API versions
  - **Alpha** – may change in incompatible way, buggy, disabled by default
  - **Beta** – schema and/or semantics of objects may change in incompatible ways, well tested, enabled by default
  - **Stable** – API will not change any time soon, production ready, enabled by default
- APIs and object schemas are organized in API groups
  - REST path - `/apis/$GROUP_NAME/$VERSION`
  - Object - `apiVersion: $GROUP_NAME/$VERSION`

NAMESPACE

- Namespaces provide scope for objects and enable division of cluster resources (e.g. resource quotas)
- Object names must be unique within namespace+kind
- Role-Based-Access-Control (RBAC) leverages namespaces for managing access rights
- Namespaces are part of the cluster internal DNS hierarchy `<service-name>.<namespace-name>.svc.cluster.local`
- Not all objects are in a namespace

```
$ kubectl get namespaces
NAME          STATUS    AGE
default       Active    1d
kube-system   Active    1d
kube-public   Active    1d

$ kubectl -n <namespace> get pods

# In a namespace
$ kubectl api-resources --namespaced=true

# Not in a namespace
$ kubectl api-resources --namespaced=false
```

LABEL

- Labels are key/value pairs that are attached to objects (e.g. Pods)
- Identifying attributes that are meaningful and relevant to users
- Used to organize and select subsets of objects
- Attached to objects at creation time and subsequently added and modified at any time

```
metadata:
  labels:
    key1: value1
    key2: value2
```

```
"release" : "stable", "release" : "canary", ...
"environment" : "dev", "environment" : "qa", "environment" : "production"
"tier" : "frontend", "tier" : "backend", "tier" : "middleware"
"partition" : "customer-a", "partition" : "customer-b", ...
"track" : "daily", "track" : "weekly"
```

LABEL SELECTOR

- Label selector is the core grouping primitive in Kubernetes
- Currently two types of selectors are supported
  - equality-based

```
environment = production
tier != frontend
```

  - set-based

```
environment in (production, qa)
tier notin (frontend, backend)
partition
```

```
$ kubectl get pods -l environment=production,tier=frontend
$ kubectl get pods -l 'environment in (production),tier in (frontend)'
$ kubectl get pods -l 'environment in (production, qa)'
$ kubectl get pods -l 'environment,environment notin (frontend)'
```

ANNOTATION

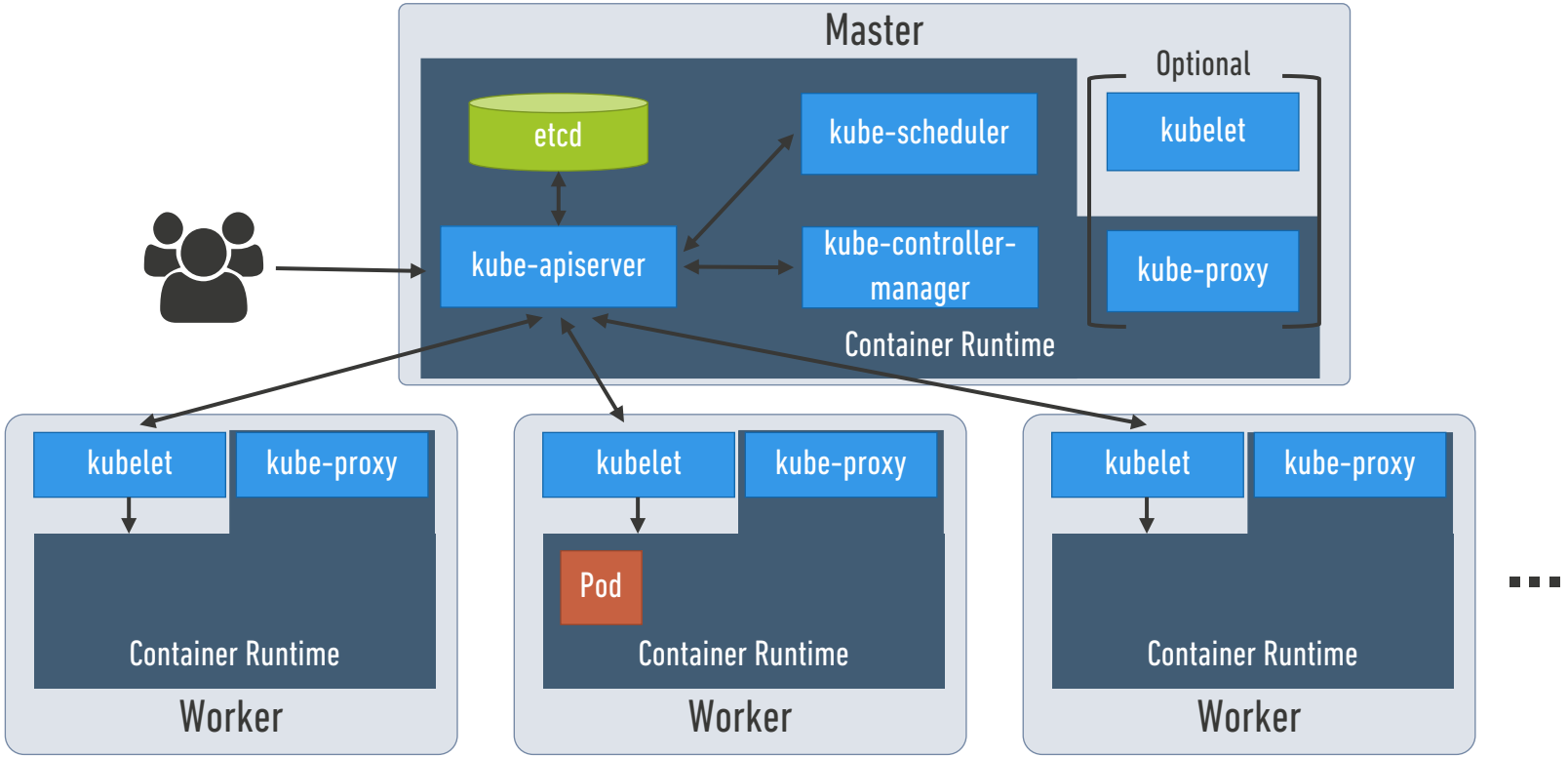
- Attach arbitrary non-identifying metadata to objects
- Annotations are not used to identify and select objects
- Metadata in an annotation can be
  - small or large
  - structured or unstructured
  - can include characters not permitted by labels
- Useful for attaching arbitrary info on an object e.g.
  - Ops Contact
  - Description
  - Debugging info

```
metadata:
  annotations:
    key1: value1
    key2: value2
```

KUBERNETES

POD

POD


Master
Optional
etcd
kube-scheduler
kubelet
kube-apiserver
kube-controller-manager
kube-proxy
Container Runtime
kubelet
kube-proxy
Container Runtime
Worker
kubelet
kube-proxy
Pod
Container Runtime
Worker
kubelet
kube-proxy
Container Runtime
Worker
...

POD

OVERVIEW

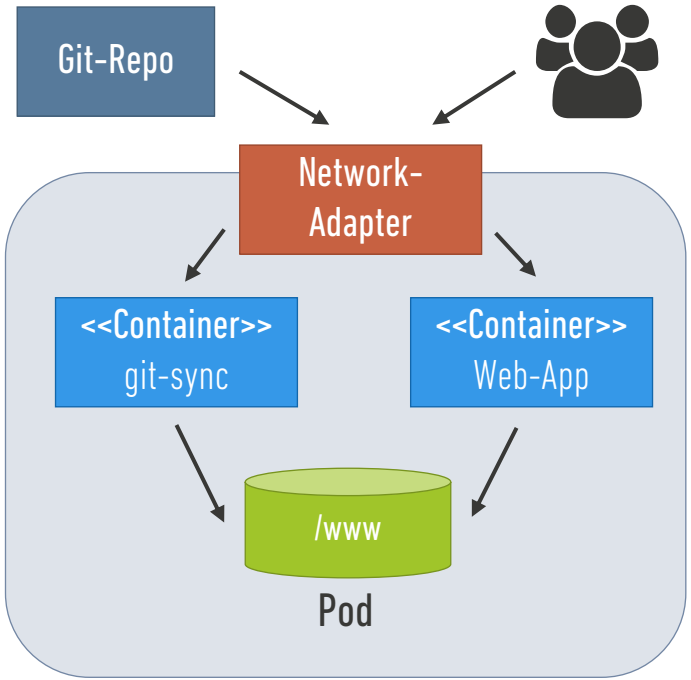
- Co-Located group of processes running with a shared context
- Models an application-specific "logical host" in a containerized environment
- Containers within a pod
  - Have access to the same IP and port space
  - Share a hostname
  - Shared volumes

→ Tightly coupled group of containers

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
    - name: myapp-container
      image: busybox
      command: ['sh', '-c', 'echo Hello Kubernetes! && sleep 3600']
```

POD

OVERVIEW



POD

OVERVIEW

- Pod is exposed as a primitive in order to facilitate
  - Easy management of co-dependent containers
  - Decoupling of controllers and services — the endpoint controller just watches pods
  - Scheduler and controller pluggability
  - Decoupling of pod lifetime from controller lifetime, such as for bootstrapping
  - Clean composition of Kubelet-level functionality with cluster-level functionality — Kubelet is effectively the "pod controller"
- Pods are ephemeral
- Users should not need to create Pods directly
  → Use higher level controllers to allow self-healing, replication and rollout management

POD

SPECIFICATION

- Composition of containers running as a single pod
  - Image
  - Image pull policy
  - Command to execute
  - Environment variables
  - Network ports
  - Hardware resources
  - Liveness-/Readiness-Probes
  - Security context
  - Termination behavior
  - Service account
  - ...

```yaml
spec:
  containers:
  - name: main
    image: k8s.gcr.io/kubernetes-dashboard-amd64:v1.10.0
    imagePullPolicy: IfNotPresent
    env:
    - name: HELLO
      value: world
    ports:
    - containerPort: 9090
      name: http
      protocol: TCP
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    dnsPolicy: ClusterFirst
    restartPolicy: Always
    schedulerName: default-scheduler
    serviceAccountName: kubernetes-dashboard
    terminationGracePeriodSeconds: 30
```

POD

LIFECYCLE

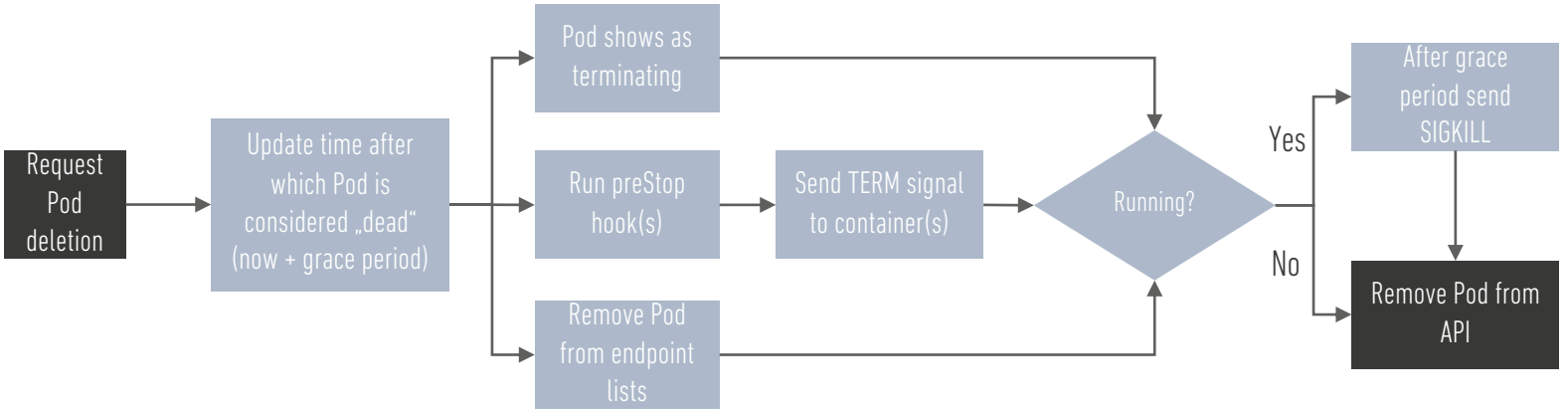
- Pod lifecycle status section
- **phase** - high-level summary of where the Pod is in its lifecycle
  - Pending
  - (Init)
  - Running
  - Succeeded
  - Failed
  - Unknown
- **conditions** - array of PodConditions through which the Pod has or has not passed

```
status:
  conditions:
    - lastProbeTime: null
      lastTransitionTime: 2018-12-17T10:20:37Z
      status: "True"
      type: Initialized
    - ...
  containerStatuses:
    - containerID: docker://f4d9a7a262ae5621f166d498a3c3bc286c92ee...
      image: k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64:1.14.10
      imageID: docker-pullable://k8s.gcr.io/k8s-dns-dnsmasq-nanny-amd64@sha256:...
      lastState: {}
      name: dnsmasq
      ready: true
      restartCount: 0
      state:
        running:
          startedAt: 2018-12-17T10:20:49Z
    - ...
  hostIP: 10.240.0.6
  phase: Running
  podIP: 10.244.0.6
  qosClass: Burstable
  startTime: 2018-12-17T10:20:37Z
```

POD

TERMINATION

- Pods represent running processes on nodes
- Important to allow those processes to gracefully terminate



POD

- Show all pods
  ```
  $ kubectl get pods
  ```
- Show details of pod
  ```
  $ kubectl describe pod <podname>
  ```
- Show logs of container in pod
  ```
  $ kubectl logs <podname>
  ```

- Install or update content of this file
  ```
  $ kubectl apply –f <filename>
  ```
- Uninstall objects defined in this file
  ```
  $ kubectl delete –f <filename>
  ```

HANDS-ON

POD

POD

1. Write a Pod yaml (https://kubernetes.io/docs/tasks/inject-data-application/define-environment-variable-container/)
   - Name → tomcat-pod
   - Image → harbor.prodyna.com/training/tomcat-sample:[id]
   - Port → 8080
   - Label → app: tomcat
   - Environment Var → STATIC_VAR = “Hello from environment”
2. Deploy yaml file (kubectl apply -f <file>)
3. Check pod and its logs („kubectl get pods“ and „kubectl logs tomcat-pod“)
4. Port-forward 8080 to localhost (kubectl port-forward) and open service in browser

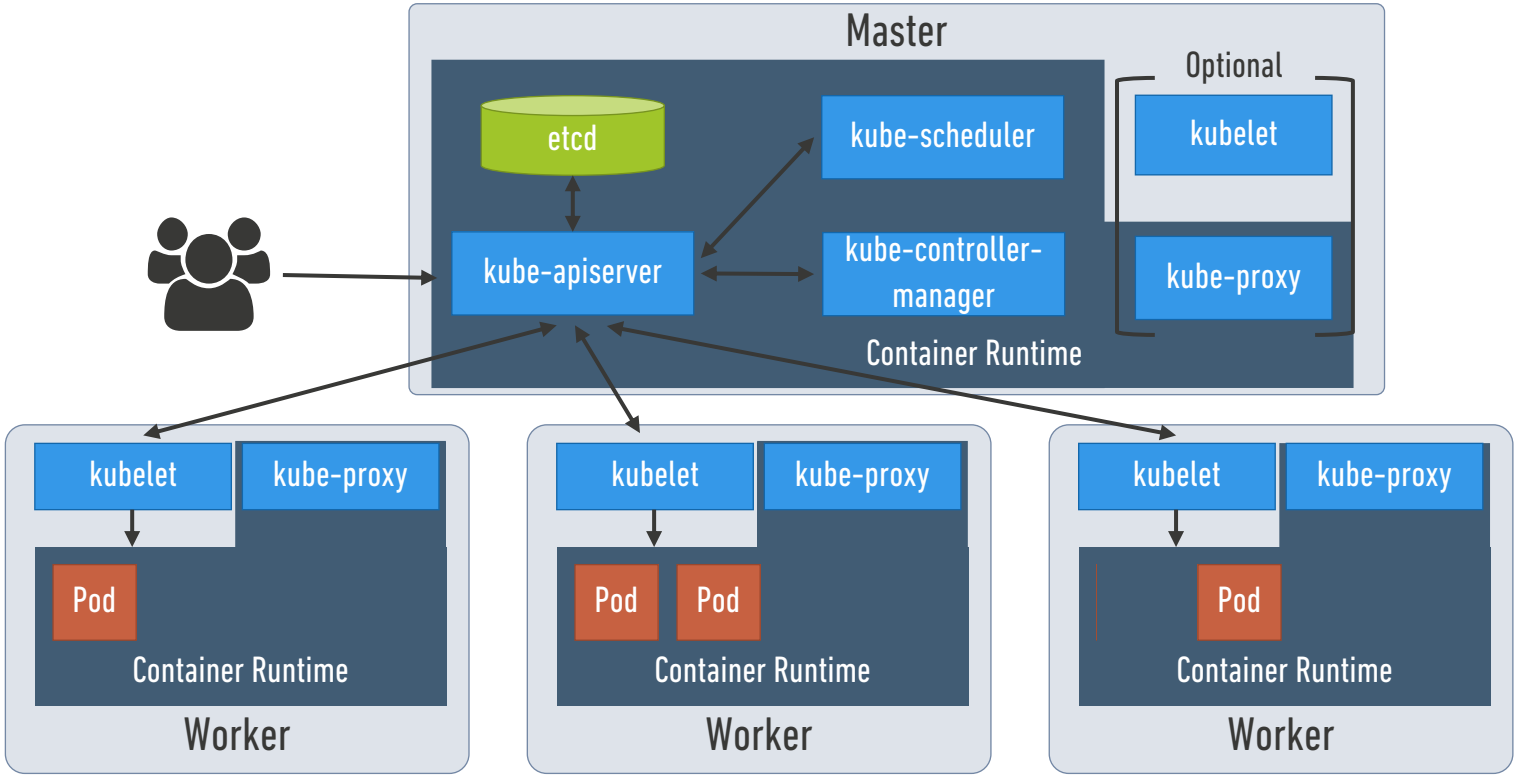
KUBERNETES

DEPLOYMENT

kubernetes

by Google

DEPLOYMENT


Master
Optional
etcd
kube-scheduler
kubelet
kube-apiserver
kube-controller-manager
kube-proxy
Container Runtime
kubelet
kube-proxy
Pod
Container Runtime
Worker
kubelet
kube-proxy
Pod
Pod
Container Runtime
Worker
kubelet
kube-proxy
Pod
Container Runtime
Worker

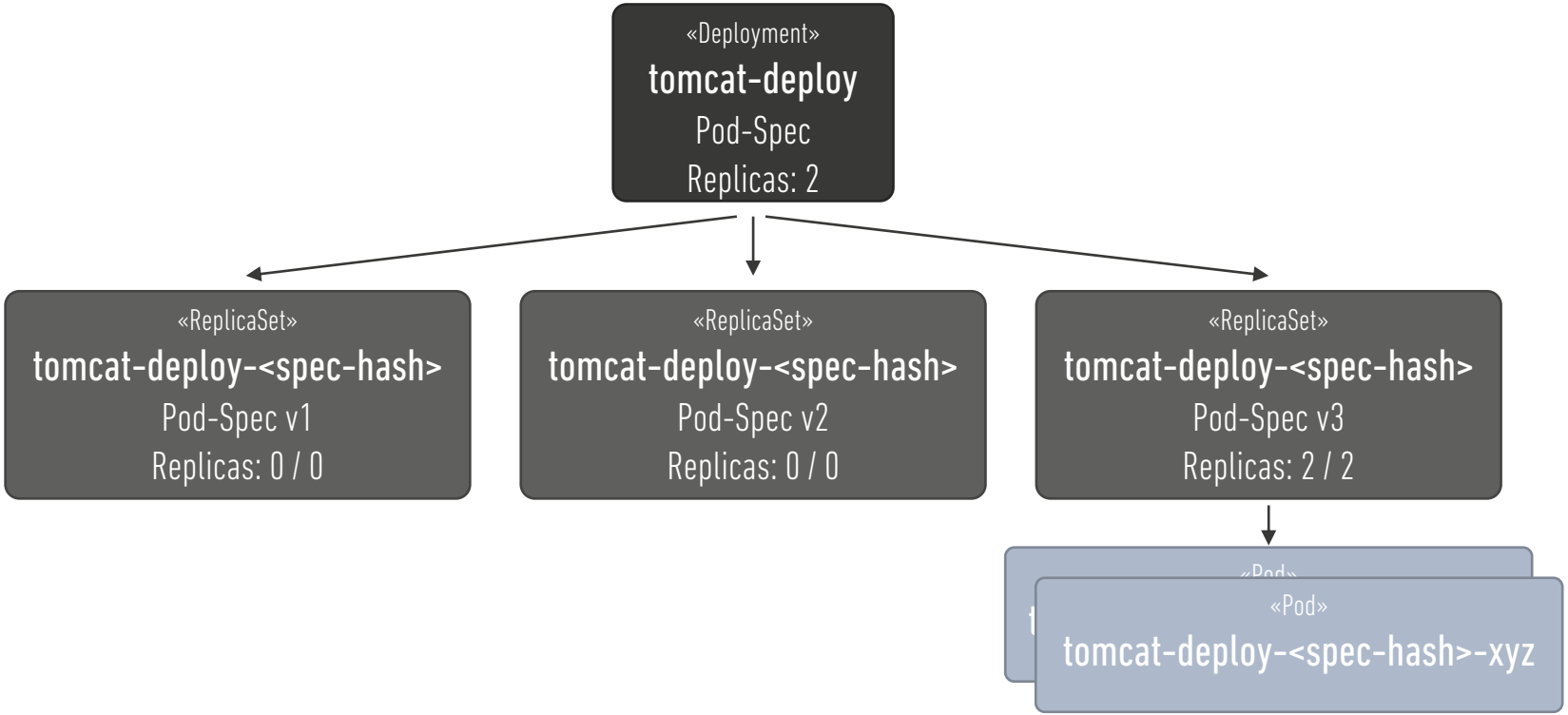
DEPLOYMENT

KUBERNETES CONCEPTS

- Provides declarative updates for Pods and ReplicaSets
- Describes the desired state and the Deployment controller will change the actual state to the desired state at a controlled rate
- Typical use-cases
  - Create a Deployment to rollout a ReplicaSet
  - Rollback to an earlier Deployment revision
  - Scale up the Deployment to facilitate more load
  - Pause the Deployment
  - Clean up older ReplicaSets

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

DEPLOYMENT


«Deployment»
tomcat-deploy
Pod-Spec
Replicas: 2
«ReplicaSet»
tomcat-deploy-<spec-hash>
Pod-Spec v1
Replicas: 0 / 0
«ReplicaSet»
tomcat-deploy-<spec-hash>
Pod-Spec v2
Replicas: 0 / 0
«ReplicaSet»
tomcat-deploy-<spec-hash>
Pod-Spec v3
Replicas: 2 / 2
«Pod»
tomcat-deploy-<spec-hash>-xyz

DEPLOYMENT

- Show all deployments
  `$ kubectl get deployment`
- Show details of deployment
  `$ kubectl describe deployment <deployment name>`
- Edit deployment on the fly
  `$ kubectl edit deployment <deployment name>`
- Show rollout history of deployment
  `$ kubectl rollout history deployment <deployment name>`

- Other relevant object: ReplicaSet
  `get, describe…`

HANDS-ON

DEPLOYMENT

DEPLOYMENT

1. Build a YAML configuration for running the tomcat-sample as a Deployment
   https://kubernetes.io/docs/concepts/workloads/controllers/deployment/
   - Name → tomcat-deploy
   - Image → harbor.prodyna.com/training/tomcat-sample:[id]
   - Replica → 1
   - Label → app: tomcat
   - Port → 8080
   - Environment Var → STATIC_VAR = “Hello from environment”

DEPLOYMENT

2. Deploy yaml file (kubectl apply -f)
3. Check Deployment
   (Describe, Logs, Port-forward and call /tomcat-sample)
4. Scale up to 2 instances
5. Change image
   → tomcat:8-jre8-alpine
6. Check rollout history
7. Rollback to initial version

KUBERNETES

OBJECT MANAGEMENT

OBJECT MANAGEMENT

- kubectl supports several different ways to create and manage Kubernetes objects

Management technique	Operates on	Recommended environment	Supported writers	Learning curve
Imperative commands	Live objects	Development	1+	Lowest
Imperative object configuration	Individual files	Production	1	Moderate
Declarative object configuration	Directory of files	Production	1+	Highest

OBJECT MANAGEMENT

IMPERATIVE COMMANDS

- User provides operations to the kubectl command as arguments or flags
- User operates directly on live objects in a cluster

```
$ kubectl run nginx --image nginx
$ kubectl create deployment nginx --image nginx
```

+ Simple, easy to learn and easy to remember
+ Require only a single step to make changes to the cluster

– No integrate with change review processes
– No audit trail associated with changes
– No source of records except for what is live
– Provides no template for creating new objects

OBJECT MANAGEMENT

IMPERATIVE OBJECT CONFIGURATION

- kubectl command specifies the operation (create, replace, etc.), optional flags and at least one file name
- File specified must contain a full definition of the object in YAML or JSON format

```
$ kubectl create -f nginx.yaml
$ kubectl delete -f nginx.yaml -f redis.yaml
$ kubectl replace -f nginx.yaml
```

+ Object configuration can be stored in a source control system
+ Object configuration can integrate with processes such as reviewing changes before push and audit trails
+ Provides templates for creating new objects

– Requires basic understanding of the object schema
– Requires additional step of writing a YAML file
– Works best on files, not directories
– Updates to live objects must be reflected in configuration files

OBJECT MANAGEMENT

DECLARATIVE OBJECT CONFIGURATION

- User operates on object configuration files stored locally
- User does **not** define the operations to be taken on the files
- Create, update, and delete operations are automatically detected per-object by kubectl

```
$ kubectl diff -R -f configs/
$ kubectl apply -R -f configs/
```

+ Changes made directly to live objects are retained, even if they are not merged back into the configuration files
+ Better support for operating on directories and automatically detecting operation types (create, patch, delete) per-object

– Harder to debug and understand results when they are unexpected
– Partial updates using diffs create complex merge and patch operations

KUBERNETES

POD CONFIGURATION

POD CONFIGURATION

RESOURCE DEFINITION

- CPU and memory are each a resource type
- A resource type has a base unit
  - **CPU** - specified in units of cores
  - **Memory** - specified in units of bytes
- One CPU, in Kubernetes, is equivalent to:
  - 1 AWS / GCP / Azure / IBM vCPU
  - 1 Hyperthread on a bare-metal Intel processor with Hyperthreading
- CPU fractional requests are allowed (e.g. 0.1 or 100m)
- Memory can be expressed as
  - Plain integer (e.g. 128974848 or 129e6)
  - Fixed-point integer suffixes: E, P, T, G, M, K (129M)
  - Power-of-two equivalents: Ei, Pi, Ti, Gi, Mi, Ki (123Mi)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: resource-demo
spec:
  containers:
  - name: resource-demo
    image: polinux/stress
    resources:
      limits:
        memory: "200Mi"
        cpu: "1"
      requests:
        memory: "100Mi"
        cpu: "0.5"
```

HANDS-ON

RESOURCE DEFINITION

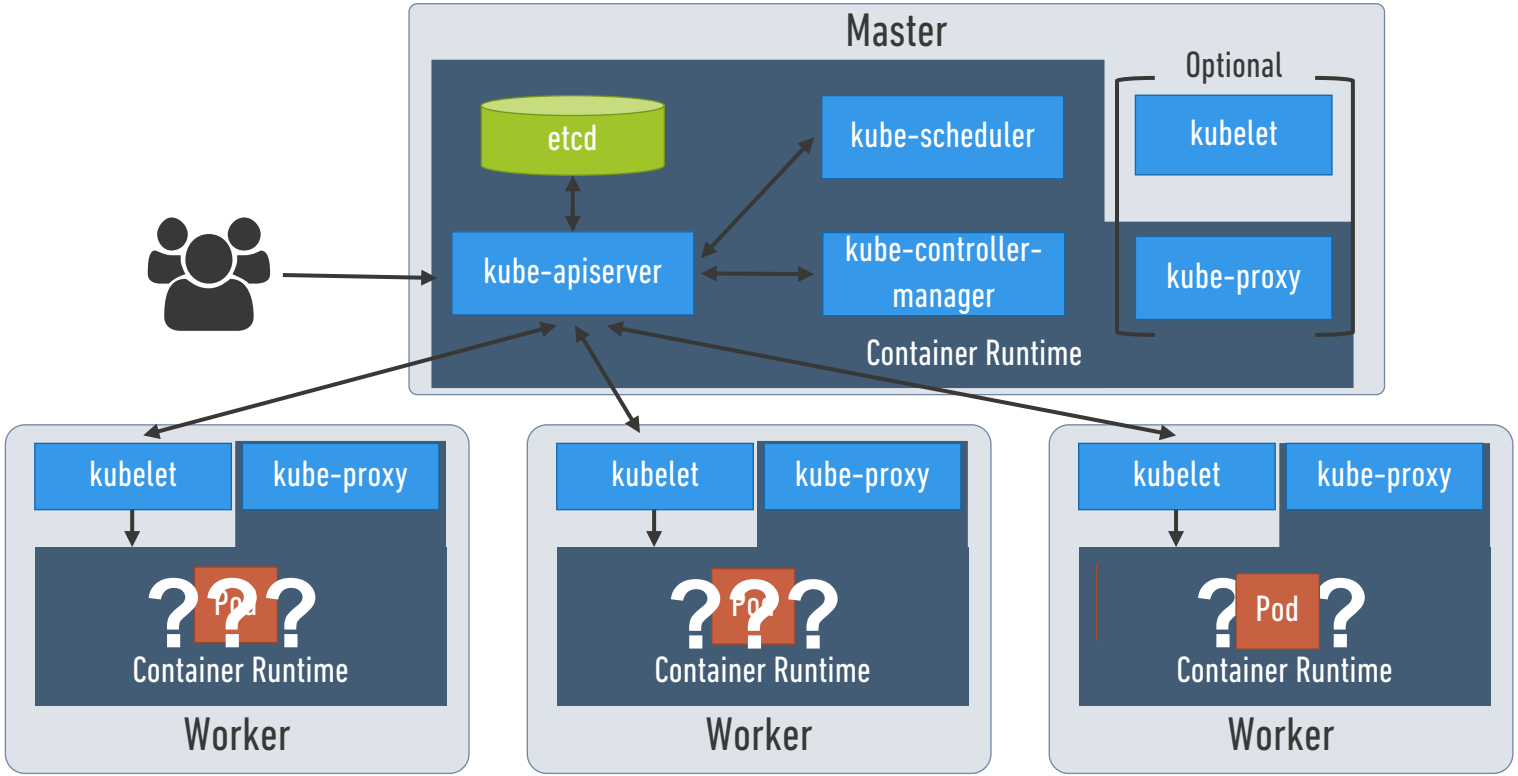
RESOURCE DEFINITION

1. Update tomcat-sample deployment
   a. Resources → RAM - Request: 16Gi
   b. Check pod status
2. Update tomcat-sample deployment
   a. Resources → RAM - Request: 512Mi Limit: 1Gi
      CPU - Request: 0.5 Limit: 1
3. Check pod status
4. Look at the node status to see resource usage

KUBERNETES

SCHEDULING – PART 1: BASICS

SCHEDULING


Master
Optional
etcd
kube-scheduler
kubelet
kube-apiserver
kube-controller-manager
kube-proxy
Container Runtime
kubelet
kube-proxy
Pod
Container Runtime
Worker
kubelet
kube-proxy
Pod
Container Runtime
Worker
kubelet
kube-proxy
Pod
Container Runtime
Worker

SCHEDULING

NODES

- Sum of all resource requests must be available on node
  (Note: Although actual memory or CPU resource usage on nodes is very low, the scheduler still refuses to place a Pod on a node if the capacity check fails)
- Simple node selection based on labels
- Nodes come pre-populated with a standard set of labels
  - kubernetes.io/hostname
  - failure-domain.beta.kubernetes.io/zone
  - failure-domain.beta.kubernetes.io/region
  - beta.kubernetes.io/instance-type
  - beta.kubernetes.io/os
  - beta.kubernetes.io/arch
- Additional labels should be prefixed with `node-restriction.kubernetes.io/`
  - Prevents kubelet from setting its own restrictions
  - Requires using Node authorizer and enabled “NodeRestriction admission plugin”

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx
  nodeSelector:
    beta.kubernetes.io/os: linux
```

HANDS-ON

SCHEDULING - NODESELECTOR

SCHEDULING

1. Check nodes for label “agentpool”
   ```
   $ kubectl get nodes -L agentpool
   ```
2. Always schedule „tomcat-sample“ deployment to „app“ node pool using nodeSelector
   a. scale up, observe pod creation, scale down

KUBERNETES

SCHEDULING – PART 2: AFFINITIES

SCHEDULING

POD CONFIGURATION

- Affinity and anti-affinity greatly expands the types of constraints you can express
  - Language is more expressive (not just “AND of exact match”)
  - Can indicate that the rule is “soft” or “preference” rather than a hard requirement
  - Can constrain against labels on other pods running on the node (inter-pod affinity/anti-affinity)
- Three types of affinity
  - nodeAffinity
  - podAffinity
  - podAntiAffinity
- Two types of node/pod affinity
  - requiredDuringSchedulingIgnoredDuringExecution
  - preferredDuringSchedulingIgnoredDuringExecution

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    nodeAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        nodeSelectorTerms:
        - matchExpressions:
          - key: kubernetes.io/e2e-az-name
            operator: In
            values:
            - e2e-az1
            - e2e-az2
      preferredDuringSchedulingIgnoredDuringExecution:
      - weight: 1
        preference:
          matchExpressions:
          - key: another-node-label-key
            operator: In
            values:
            - another-node-label-value
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

SCHEDULING

POD CONFIGURATION

PodAffinity / PodAntiAffinity:

- When do pods count as „being together“?
  - Same node?
  - Same server rack?
  - Same availability zone?
  - Same group of nodes defines by other criteria?

- TopologyKey chooses a label that identifies node groups
- Nodes with the same label value belong to the same group

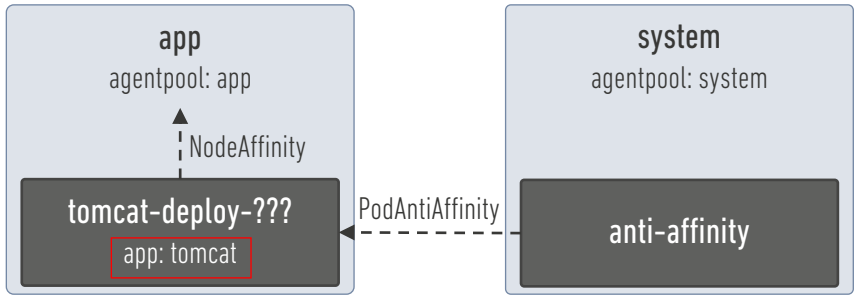
```yaml
apiVersion: v1
kind: Pod
metadata:
  name: with-node-affinity
spec:
  affinity:
    podAffinity:
      requiredDuringSchedulingIgnoredDuringExecution:
        - labelSelector:
            matchExpressions:
            - key: app
              operator: In
              values:
              - database
          topologyKey: kubernetes.io/e2e-az-name
  containers:
  - name: with-node-affinity
    image: k8s.gcr.io/pause:2.0
```

HANDS-ON

SCHEDULING

SCHEDULING

1. Replace nodeSelector with nodeAffinity and update pod
   a. requiredDuringSchedulingIgnoredDuringExecution
   b. scale up, observe pod creation
2. Create deployment that is never scheduled on the same node as tomcat-sample
   a. Name → anti-affinity
   b. Image → k8s.gcr.io/pause:2.0
   c. podAntiAffinity
      a. requiredDuringSchedulingIgnoredDuringExecution
      b. topologyKey: kubernetes.io/hostname



KUBERNETES

SCHEDULING – PART 3: TAINTS & TOLERATIONS

SCHEDULING

POD CONFIGURATION

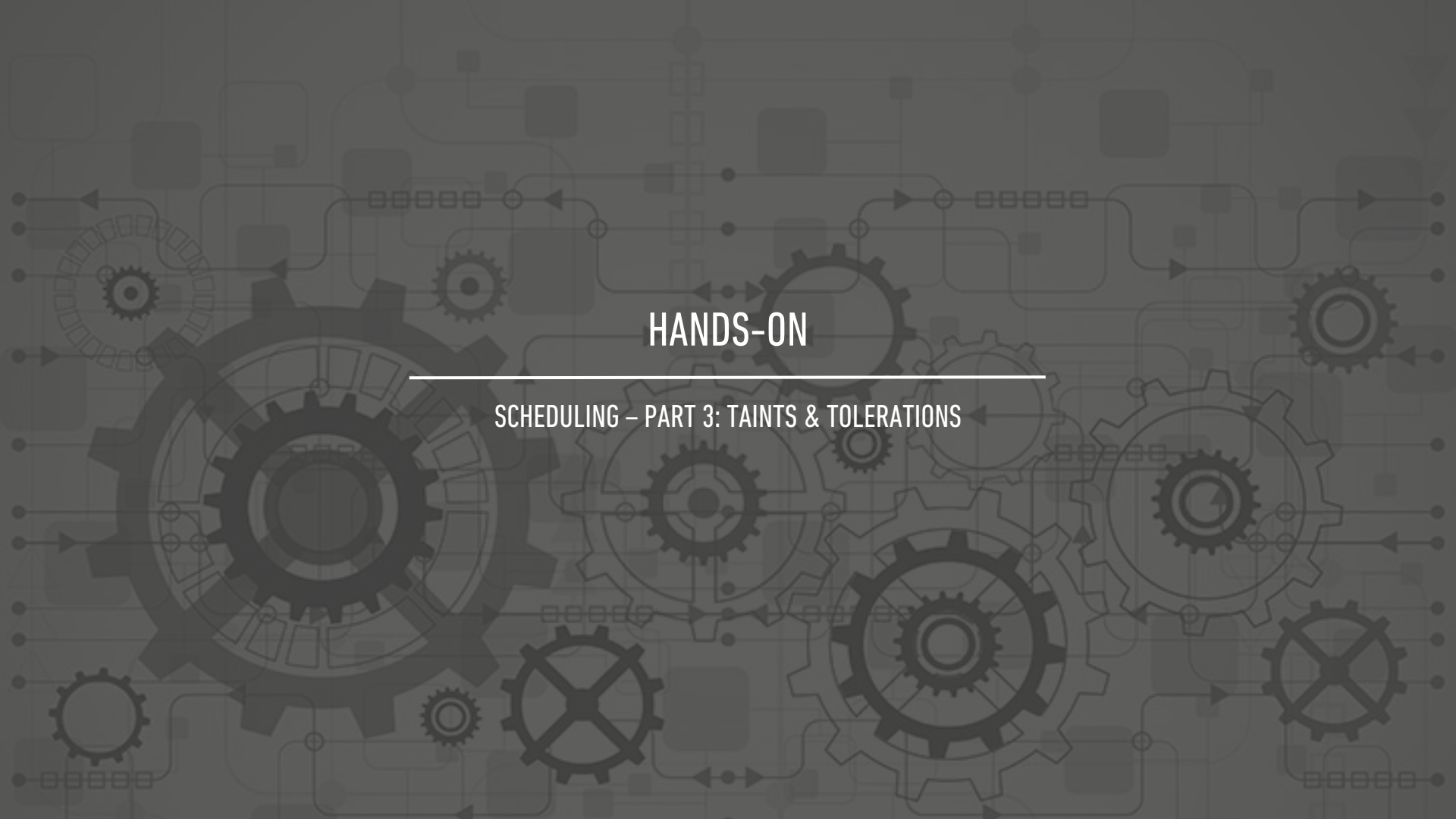
- Taints and Tolerations allow a node to repel a set of pods
- One or more taints can be applied to a node
- Taints are applied to nodes and can have different effects
  - NoSchedule
  - PreferNoSchedule
  - NoExecute
- Tolerations are applied to pods, allow them to ignore taints

```
$ kubectl taint nodes node1 disk=hdd:NoSchedule
$ kubectl taint nodes node1 disk:NoSchedule-

tolerations:
- key: "network"
  operator: "Equal"
  value: "slow"
  effect: "NoSchedule"

tolerations:
- key: "network"
  operator: "Exists"
  effect: "NoSchedule"

tolerations:
- key: "network"
  operator: "Equal"
  value: "slow"
  effect: "NoExecute"
  tolerationSeconds: 3600
```



HANDS-ON

SCHEDULING – PART 3: TAINTS & TOLERATIONS

SCHEDULING

1. Clean up deployments/pods from previous exercises
2. Taint app node
   a. foo=bar:NoSchedule
3. Recreate deployment tomcat-deploy from previous exercise (with nodeAffinity)
4. Observe where tomcat-deploy-??? pods get recreated
5. Add toleration to tomcat-deploy deployment and apply
6. Remove taint from app node

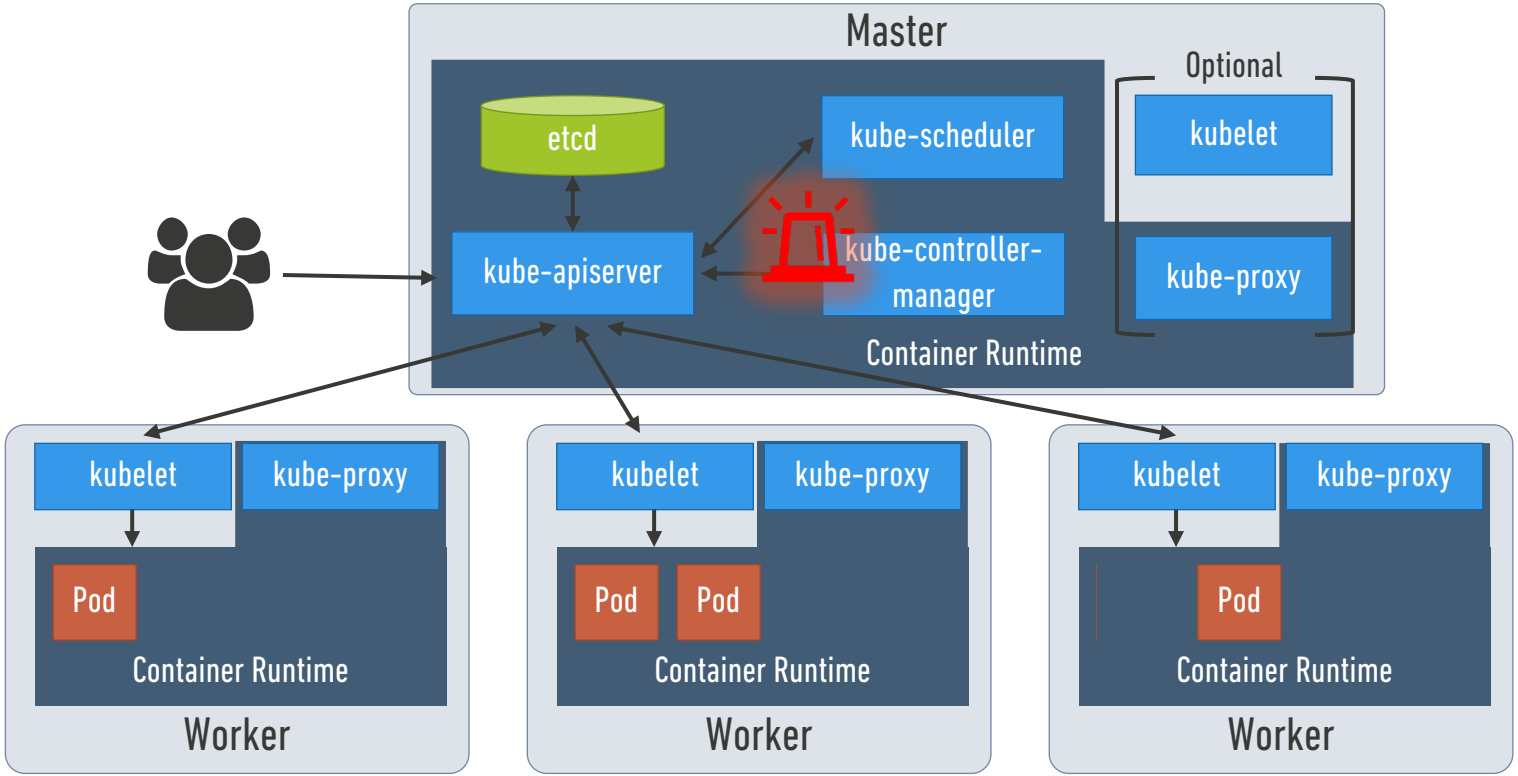
KUBERNETES

CONTAINER PROBES

kubernetes

by Google

CONTAINER PROBES



POD CONFIGURATION

CONTAINER PROBES

- Diagnostic performed periodically by the kubelet on a container
- kubelet calls a handler implemented by the container
  - **ExecAction** - Executes a specified command inside the container
  - **TCPSocketAction** - TCP check against the container's IP address on a specified port
  - **HTTPGetAction** - HTTP Get against the container's IP address on a specified port and path
- Each probe has three possible results
  - **Success** - The container passed the diagnostic
  - **Failure** - The container failed the diagnostic
  - **Unknown** - The diagnostic failed, so no action should be taken
- Three kinds of probes on running containers
  - **startupProbe –** Indicates successful pod start (deactivated after first success)
  - **livenessProbe** - Indicates if the container is running (activated after startup probe)
  - **readinessProbe** - Indicates if the Container is ready to service requests

```yaml
livenessProbe:
  failureThreshold: 5
  httpGet:
    path: /healthz-kubedns
    port: 8080
    scheme: HTTP
  initialDelaySeconds: 60
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 5

readinessProbe:
  failureThreshold: 3
  httpGet:
    path: /readiness
    port: 8081
    scheme: HTTP
  initialDelaySeconds: 30
  periodSeconds: 10
  successThreshold: 1
  timeoutSeconds: 5
```

HANDS-ON

CONTAINER PROBES

CONTAINER PROBES

1. Enhance Tomcat deployment with liveness- and readinessProbe using execAction
   a. wget --spider http://localhost:8080/
2. ~~Change liveness- and readinessProbe to TCPSocketAction~~
3. Change liveness- and readinessProbe to HTTPGetAction
   a. Cause the liveness probe to fail, observe pod restarts

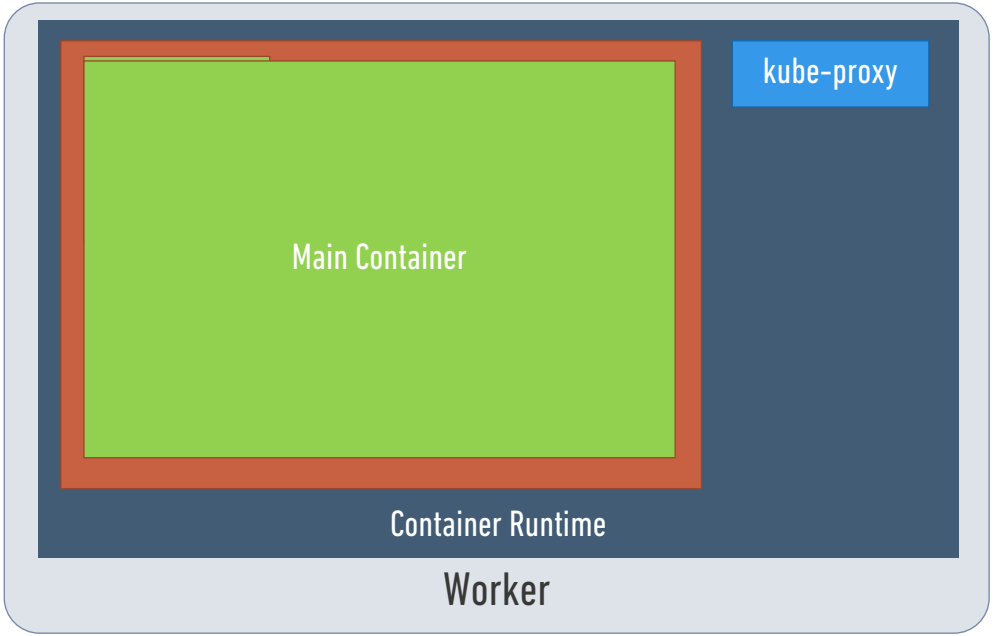
KUBERNETES

INIT CONTAINERS

kubernetes

by Google

INIT CONTAINER



POD CONFIGURATION

INIT-CONTAINER

- Pods can have one or more init-containers
- Init-containers are run before the app containers are started
- Init-containers are exactly like regular containers, except:
  - They always run to completion
  - Each one must complete successfully before the next one is started
- If an init-container fails for a pod, Kubernetes restarts the pod (see restartPolicy)

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
spec:
  initContainers:
  - name: init-mydb
    image: busybox
    command:
    - 'sh'
    - '-c'
    - 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;'
  containers:
  - name: myapp-container
    image: busybox
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
```

HANDS-ON

INIT-CONTAINER

INIT-CONTAINER

1. Deploy kubernetes-training/k8s/init-container-sample.yaml
2. Check pod status and logs
3. Fix the pod and redeploy! ☺

KUBERNETES

CONFIG MAP

kubernetes

by Google

CONFIG MAP

- Allows to decouple configuration artifacts from image content
- Data consists of one or more key/value pairs
- Value can be a complex file
- Data can be referenced by pod:
  1. Environment variable
  2. Mount to file system (key is file name)

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: some-config
data:
  my-key: my-value
  ui.properties: |
    color.good=purple
    color.bad=yellow
    allow.textmode=true
    how.nice.to.look=fairlyNice
```

HANDS-ON

CONFIG MAP

CONFIG MAP

1. Create ConfigMap
   - Name → tomcat-cm
   - Content
     - color: red
     - sample.txt: |
       Some
       multiline
       text!
2. Enhance Tomcat deployment
   - Set value of color to environment variable “COLOR”
   - Mount to /tomcat-cfg
3. Check result

```
$ kubectl exec tomcat-deploy-??? -- env
$ kubectl exec tomcat-deploy-??? -- cat /tomcat-cfg/sample.txt
```

AGENDA

DAY 02

1. Workload definition
2. Configuration
   - ConfigMap, Secret
3. Network
   - Services, Ingress
4. Controllers
   - Deployments, StatefulSets, DaemonSet
5. Persistence
   - Volume, PersistentVolume, PersistentVolumeClaim, StorageClass

KUBERNETES

SECRET

SECRET

- Intended to hold sensitive information
  - Passwords
  - OAuth tokens
  - ssh keys
  - …
- Similar to ConfigMap but values are Base64 encoded (not encrypted)
- Secrets can be referenced by pod:
  1. Environment variable
  2. Mount to file system (key is file name)

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: mysecret
type: Opaque
data:
  username: YWRtaW4=
  password: MWYyZDFlMmU2N2Rm
```

HANDS-ON

SECRET

SECRET

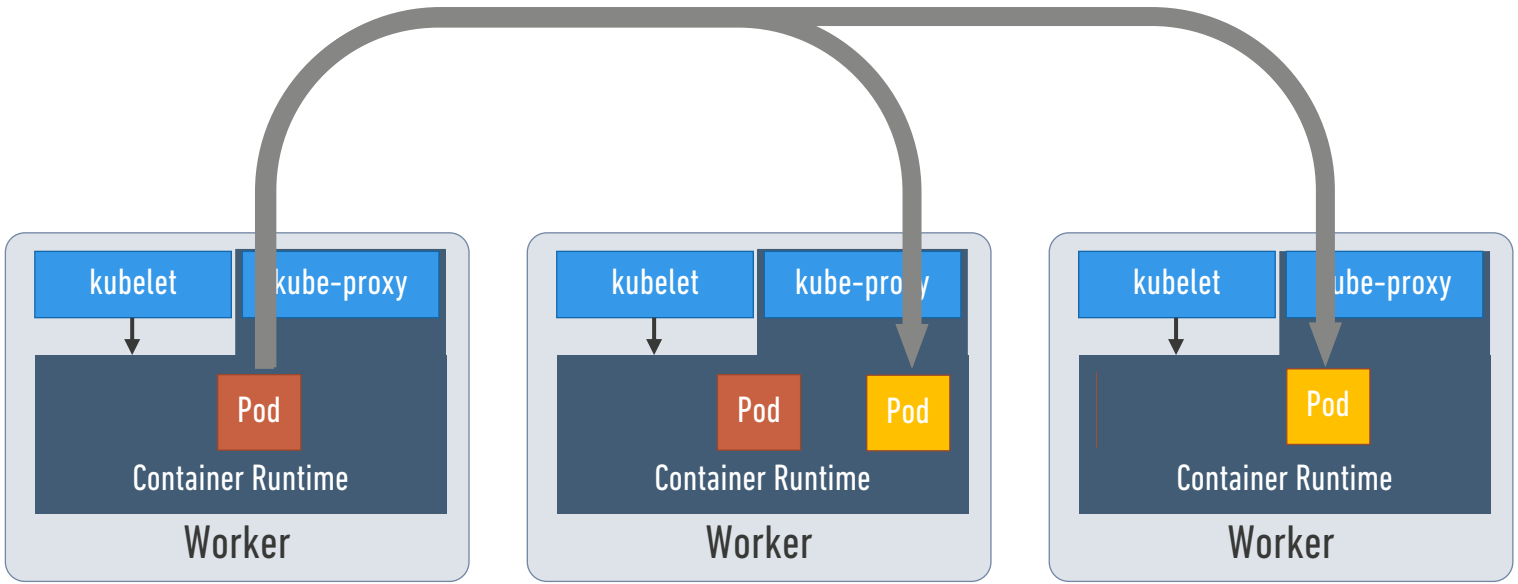
1. Create a Secret
   - Name → tomcat-secret
   - Content
     - password: secret
2. Enhance Tomcat deployment
   - Set value of "password" to environment variable "PASSWORD"
   - Mount Secret to /tomcat-secret
3. Check result

```
$ kubectl exec tomcat-deploy-??? -- env
$ kubectl exec tomcat-deploy-??? -- cat /tomcat-secret/password
```

KUBERNETES

SERVICE

SERVICE


kubelet
kube-proxy
Pod
Container Runtime
Worker
kubelet
kube-proxy
Pod
Pod
Container Runtime
Worker
kubelet
kube-proxy
Pod
Container Runtime
Worker

SERVICE

- Pods usually should not connect directly to each other
  - Pod IPs can change as pods are ephemeral
  - Pods don't know about each other's replicas etc.
  - Direct access would bypass load balancing

- Services provide an abstraction to solve these problems
  - Define a set of pods
  - DNS
  - Load Balancing

SERVICE

- Set of Pods is defined by label selector
- Services can be reliably accessed from within the Kubernetes cluster and from the outside world (kube-proxy does the routing and load-balancing)
- Service can be resolved via DNS
  `<service-name>.<namespace>.svc.cluster.local`
- Two types of Kubernetes objects
  - `service (svc)`
  - `endpoint`
- Services/Endpoints get updated when labels or readiness changes

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 9376
  - name: https
    protocol: TCP
    port: 443
    targetPort: 9377
```

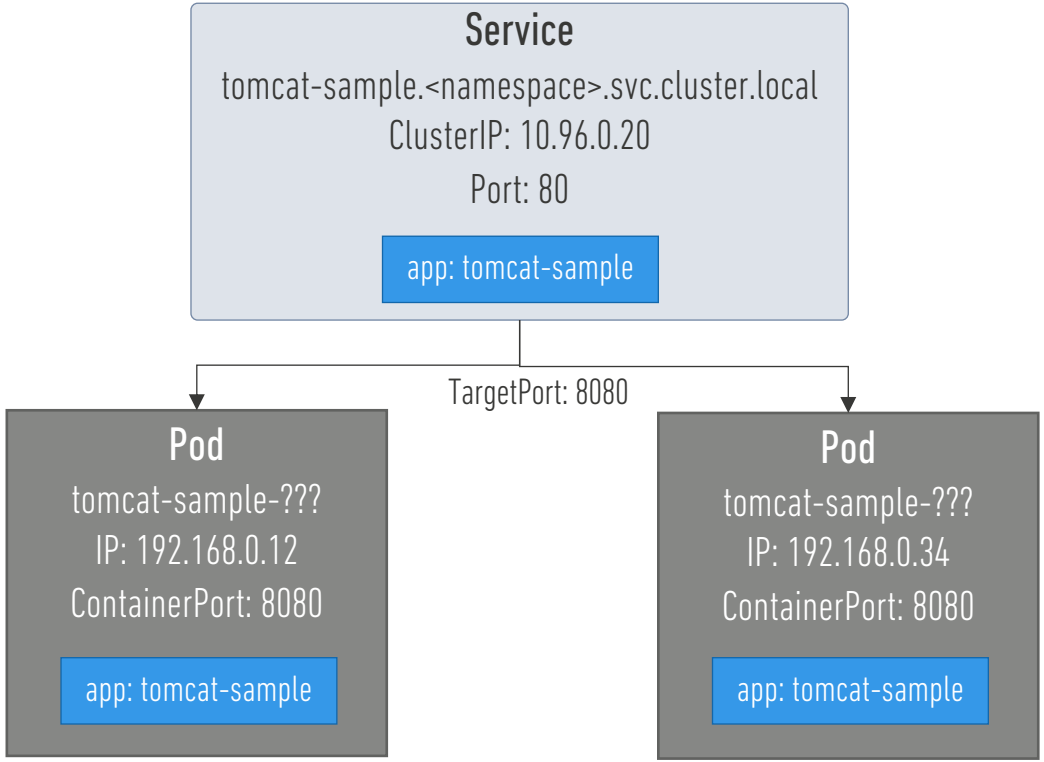
SERVICE

- Four different types
  - **ClusterIP** – Exposes the service on a cluster-internal IP
  - **NodePort** – Exposes the service on each Node's IP at a static port
  - **LoadBalancer** - Exposes the service externally using a cloud provider's load balancer
  - **ExternalName** - Maps the service to the contents of the externalName field by returning a CNAME record with its value
- Services are configurations, not processes
  - Manifested in IP tables of nodes
  - Continuously checked and updated by Kubernetes (Network Plugins)

```yaml
kind: Service
apiVersion: v1
metadata:
  name: my-service
spec:
  selector:
    app: MyApp
  ports:
  - name: http
    protocol: TCP
    port: 80
    targetPort: 9376
  - name: https
    protocol: TCP
    port: 443
    targetPort: 9377
```

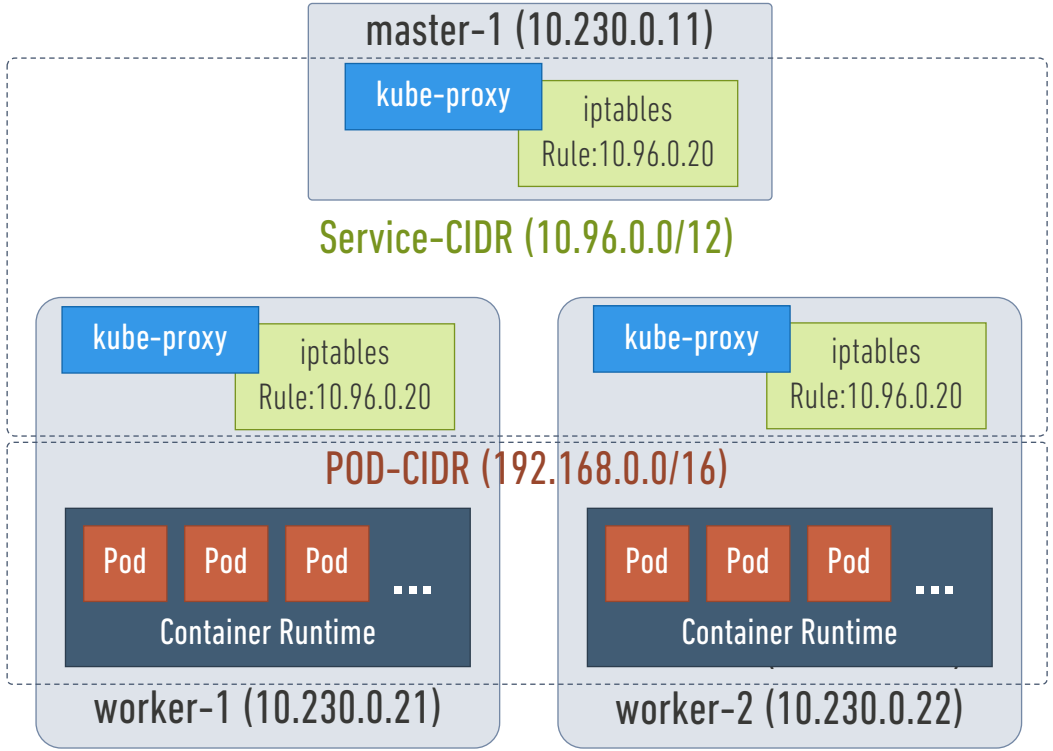
SERVICE

CLUSTER-IP



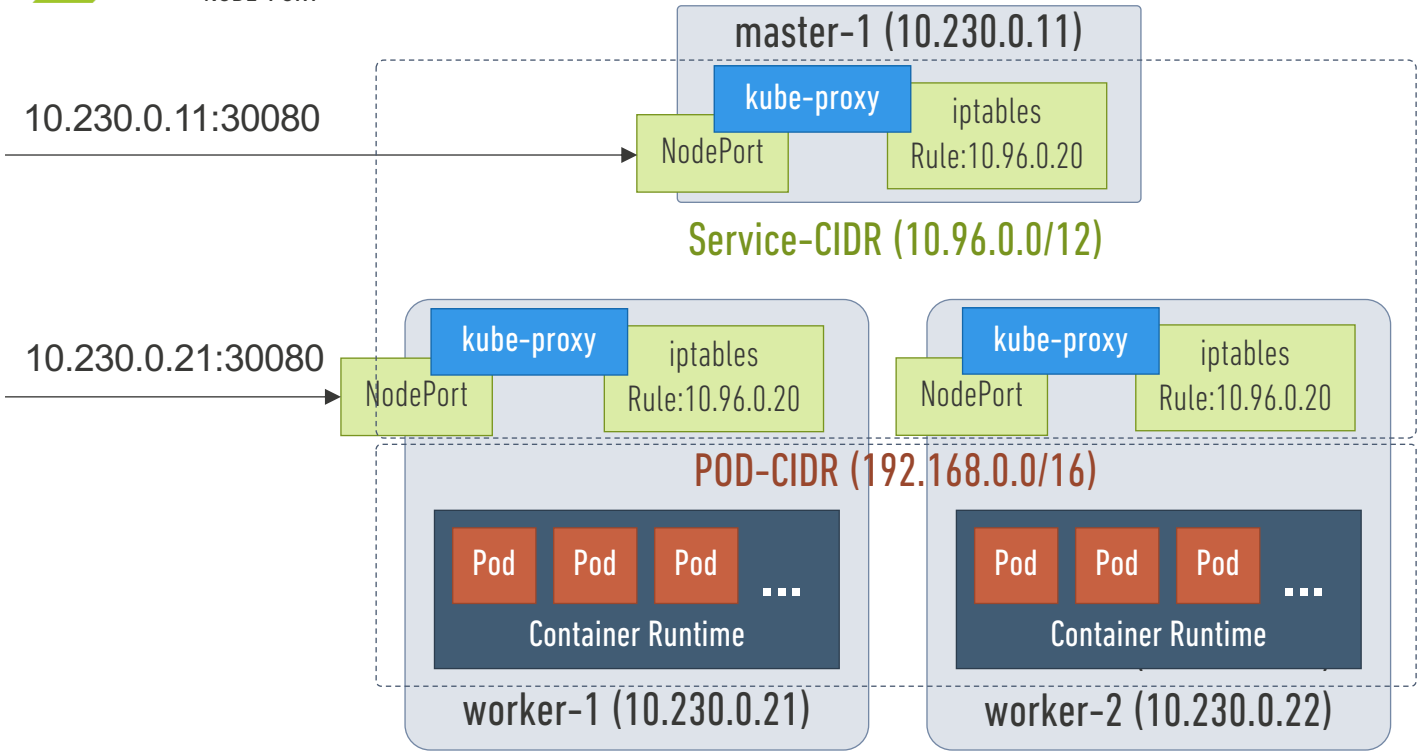
SERVICE

CLUSTER-IP



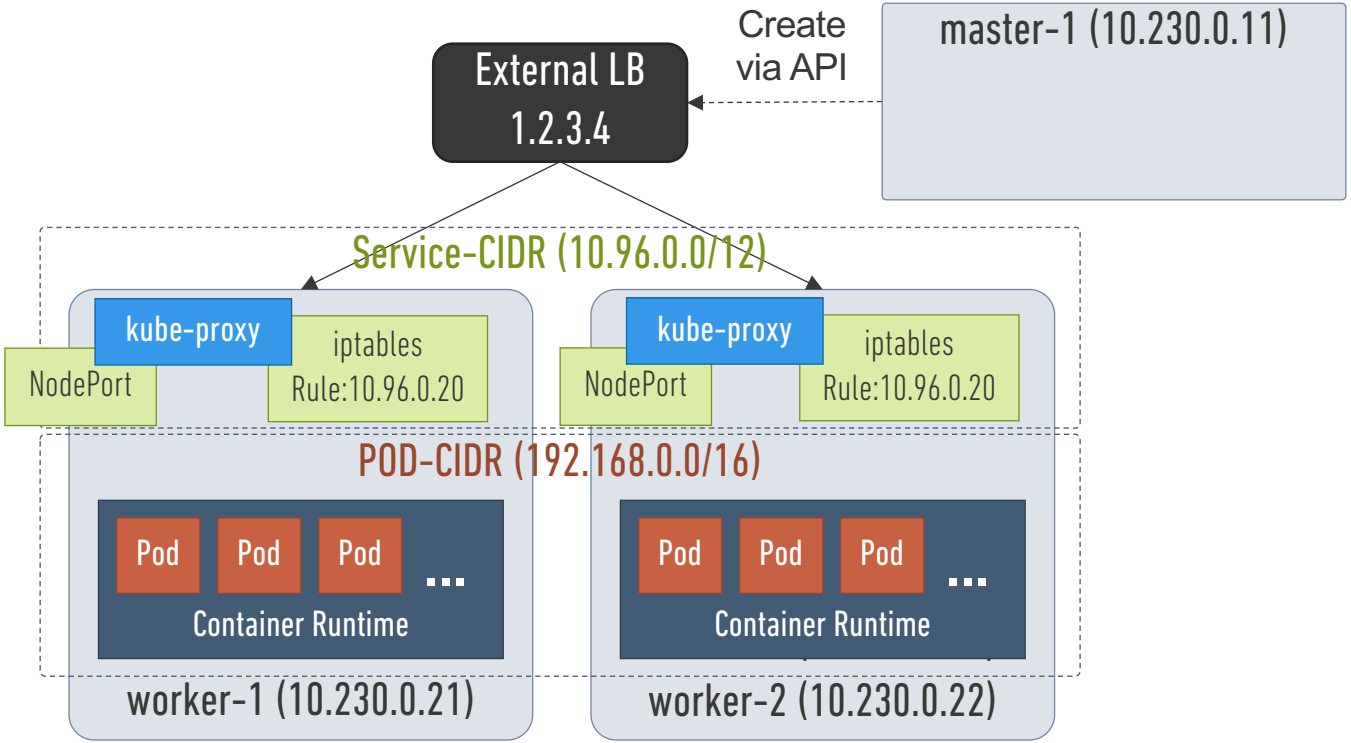
SERVICE

NODE-PORT



SERVICE

LOAD-BALANCER



HANDS-ON

SERVICE

SERVICE

1. Build a YAML configuration of a Service for accessing tomcat-sample-deployment (https://kubernetes.io/docs/concepts/services-networking/service/)
   a. Name → tomcat-svc
   b. Selector → app: tomcat
   c. Type → LoadBalancer
   d. Ports → port: 80 – targetPort: 8080
2. Deploy yaml

   ```
   $ kubectl apply -f [YAML_FILE]
   ```
3. Describe service and access via its external IP

KUBERNETES

INGRESS

kubernetes

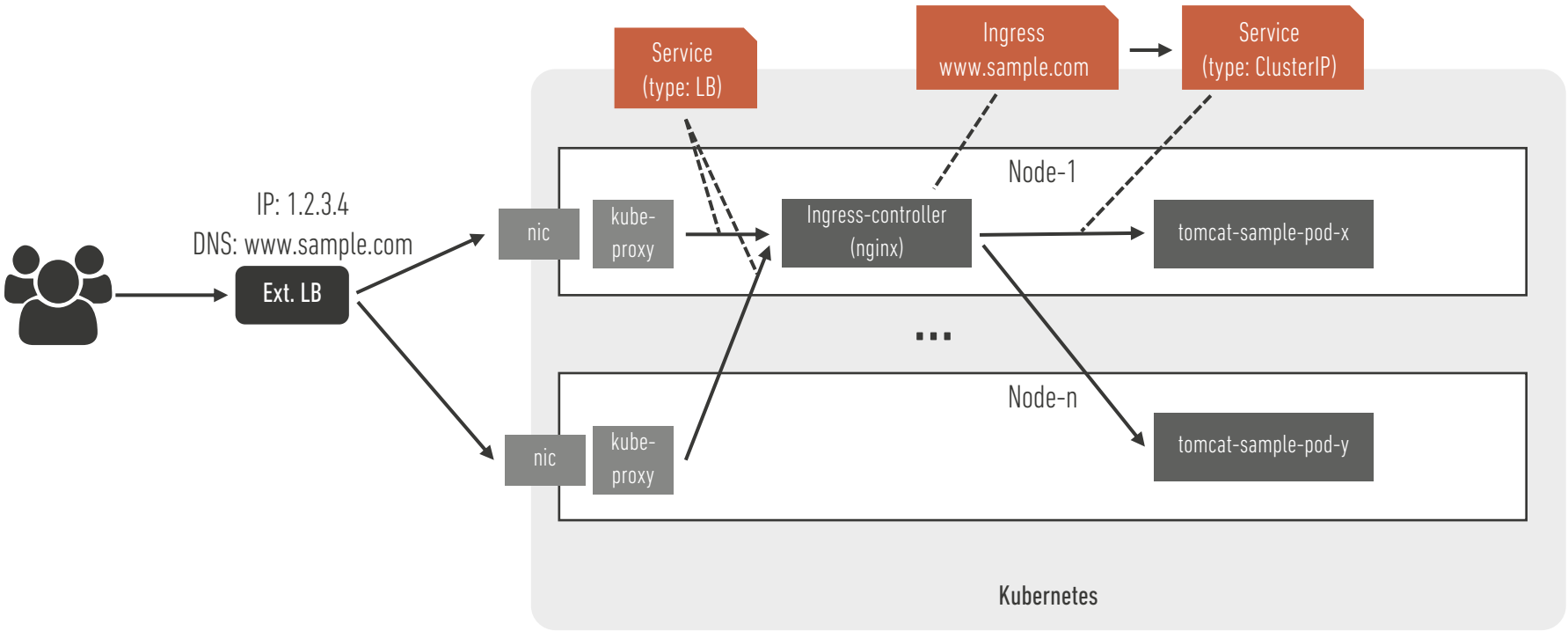
by Google

INGRESS

- Manages external access to the services in a cluster
- Services and pods have IPs only routable by the cluster network
- Ingress is a collection of rules that allow inbound connections to reach the cluster services

```yaml
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: simple-fanout-example
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  ingressClassName: nginx
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        pathType: Prefix
        backend:
          service:
            name: service1
            port:
              number: 4200
      - path: /bar
        pathType: Prefix
        backend:
          service:
            name: service2
            port:
              number: 8080
```

INGRESS


Service
(type: LB)
Ingress
www.sample.com
Service
(type: ClusterIP)
Node-1
IP: 1.2.3.4
DNS: www.sample.com
nic
kube-
proxy
Ingress-controller
(nginx)
tomcat-sample-pod-x
Ext. LB
...
Node-n
nic
kube-
proxy
tomcat-sample-pod-y
Kubernetes

HANDS-ON

INGRESS

INGRESS

We need to send requests to hostnames which need to resolve to our ingress service IP

- Nip.io
  - Dynamic DNS service
  - tomcat.<EXTERNAL-IP>.nip.io DNS lookup resolves to external IP
  - Browser sends request to external IP, server receives the above hostname
- Curl with host header:
  ```
  $ curl –H “Host: tomcat.<EXTERNAL-IP>.nip.io” ...
  ```
- Custom entry in hosts file that resolves hostnames to our desired IP

INGRESS

1. Deploy nginx ingress controller

   ```
   $ kubectl apply –f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.3.1/deploy/static/provider/cloud/deploy.yaml
   ```

2. Get external IP of Load-Balancer

   ```
   $ kubectl get svc -n ingress-nginx
   ```

3. Verify that nginx is available

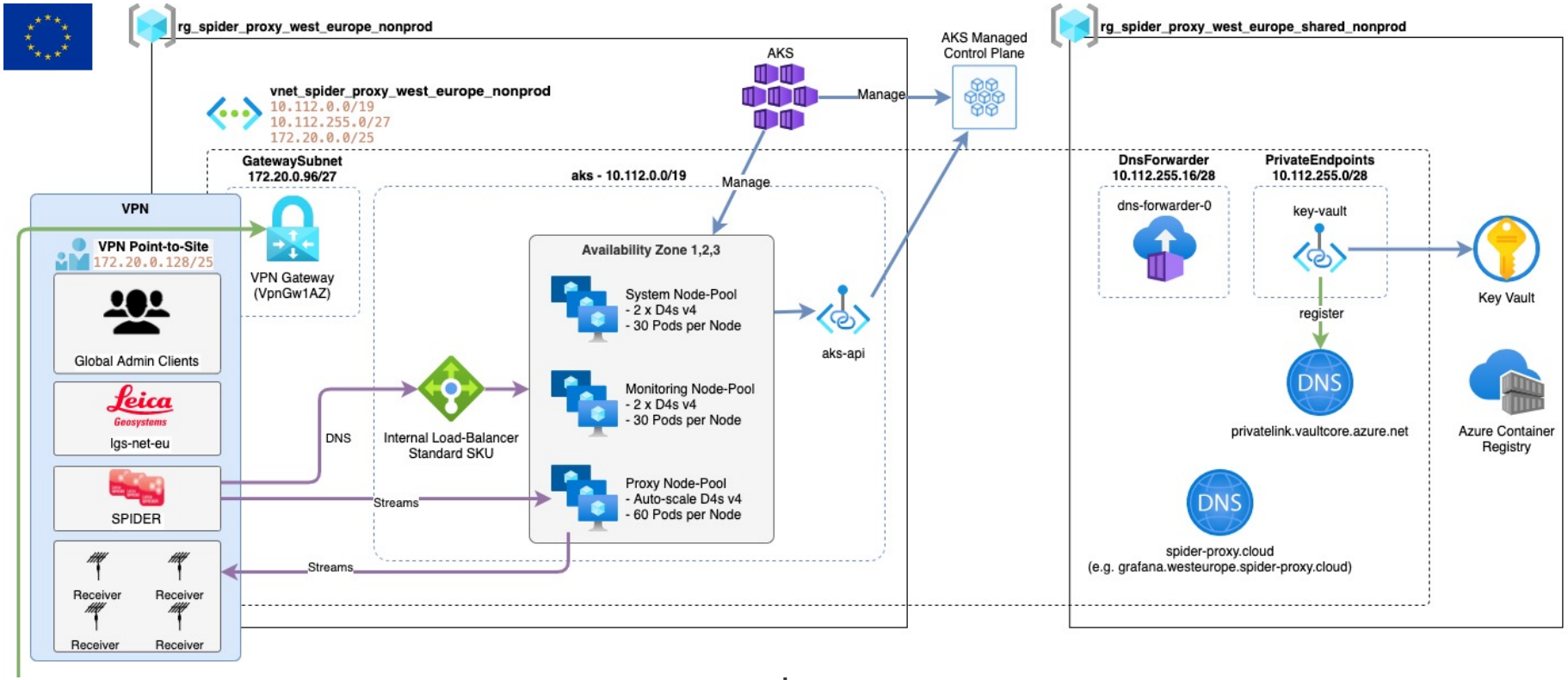
   http://tomcat.<EXTERNAL-IP>.nip.io

   Expected:

4. Create ingress for host „tomcat.<EXTERNAL-IP>.nip.io“ and route to service „tomcat-svc“
   - ingressClassName: nginx

SMARTNET APP PROXY NETWORK ARCHITECTURE



KUBERNETES

RBAC

kubernetes

by Google

(CLUSTER)ROLE

- **Role** can only be used to grant access to resources within a single namespace
- A **ClusterRole** can be used to grant the same permissions as a Role, but because they are cluster-scoped
  - cluster-scoped resources (like nodes)
  - non-resource endpoints (like "/healthz")
  - namespaced resources (like pods) across all namespaces (needed to run kubectl get pods --all-namespaces, for example)
- Many default roles available https://kubernetes.io/docs/reference/access-authn-authz/rbac/

```yaml
kind: Role
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  namespace: default
  name: pod-reader
rules:
- apiGroups: [""] # "" indicates the core API group
  resources: ["pods"]
  verbs: ["get", "watch", "list"]

---

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  # "namespace" omitted since ClusterRoles
  name: secret-reader
rules:
- apiGroups: [""]
  resources: ["secrets"]
  verbs: ["get", "watch", "list"]
```

(CLUSTER)ROLEBINDING

- Grants the permissions defined in a role to a user or set of users
- Made up of two parts
  - List of subjects (users, groups, or service accounts)
  - Reference to the role being granted
- Permissions can be granted within a namespace with a **RoleBinding**
- Permissions can be granted cluster-wide with a **ClusterRoleBinding**

```yaml
# This role binding allows "jane" to
# read pods in the "default" namespace.
kind: RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: read-pods
  namespace: default
subjects:
- kind: User
  name: jane # Name is case sensitive
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role #this must be Role or ClusterRole
  # this must match the name of the
  # Role or ClusterRole you wish to bind to
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

HANDS-ON

RBAC

RBAC

1. Create ServiceAccount yaml
   - Name: tomcat
   - imagePullSecret: harbor
2. Create Role yaml
   - Name: read-config
   - get, list, watch → configmaps, secrets
3. Create RoleBinding
   - Name: tomcat-read-config
   - Role: read-config
   - Subject Name: tomcat
   - Subject Kind: ServiceAccount

RBAC

4. Enter shell of tomcat-sample pod and call API

```
$ KUBE_TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)

$ wget --no-check-certificate --header="Authorization: Bearer $KUBE_TOKEN"
-O - https://kubernetes/api/v1/namespaces/default/configmaps
```

Expected:

```
bash-4.4# wget --no-check-certificate --header="Authorization: Bearer $KUBE_TOKEN" -O - https://kubernetes/api/v1/namespaces/default/configmaps
Connecting to kubernetes (10.96.0.1:443)
wget: server returned error: HTTP/1.1 403 Forbidden
```

5. Modify existing deployment to use “tomcat” ServiceAccountName
6. Repeat step 4

KUBERNETES

ADVANCED YAML CONFIGURATION

ADVANCED YAML CONFIGURATION

- YAML Configs can become complex and verbose
- Customization is required for
  - Staging for dev / test / qa / prod
  - Adapting own software to client deployment scenario
  - Adapting third party software to own deployment scenario
- Customization should be convenient and robust

ADVANCED YAML CONFIGURATION

DIFFERENT STRATEGIES – TEMPLATING AND PATCHING

- Templating (Helm)
  - Similar to templating in HTML
  - Very powerful
  - Complex to develop
  - Learning a new language
  - Customization limited to what the template supports
- Patching (Kustomize)
  - Makes adjustments to a base YAML config
  - Less powerful but easier to use
  - Everything can be patched
  - Uses mostly just native Kubernetes syntax

Cassandra Helm chart example excerpt:

```
{{- if .Values.podLabels }}
{{ toYaml .Values.podLabels | indent 8 }}
{{- end }}
{{- if .Values.podAnnotations }}
      annotations:
{{ toYaml .Values.podAnnotations | indent 8 }}
{{- end }}
    spec:
      {{- if .Values.schedulerName }}
      schedulerName: "{{ .Values.schedulerName }}"
      {{- end }}
      hostNetwork: {{ .Values.hostNetwork }}
{{- if .Values.selector }}
{{ toYaml .Values.selector | indent 6 }}
{{- end }}
      {{- if .Values.securityContext.enabled }}
      securityContext:
        fsGroup: {{ .Values.securityContext.fsGroup }}
        runAsUser: {{ .Values.securityContext.runAsUser
}}
      {{- end }}
{{- if .Values.affinity }}
      affinity:
{{ toYaml .Values.affinity | indent 8 }}
{{- end }}
```

Source:
https://github.com/helm/charts/blob/master/incubator/cassandra/templates/statefulset.yaml

HELM

THE PACKAGE MANAGER FOR KUBERNETES

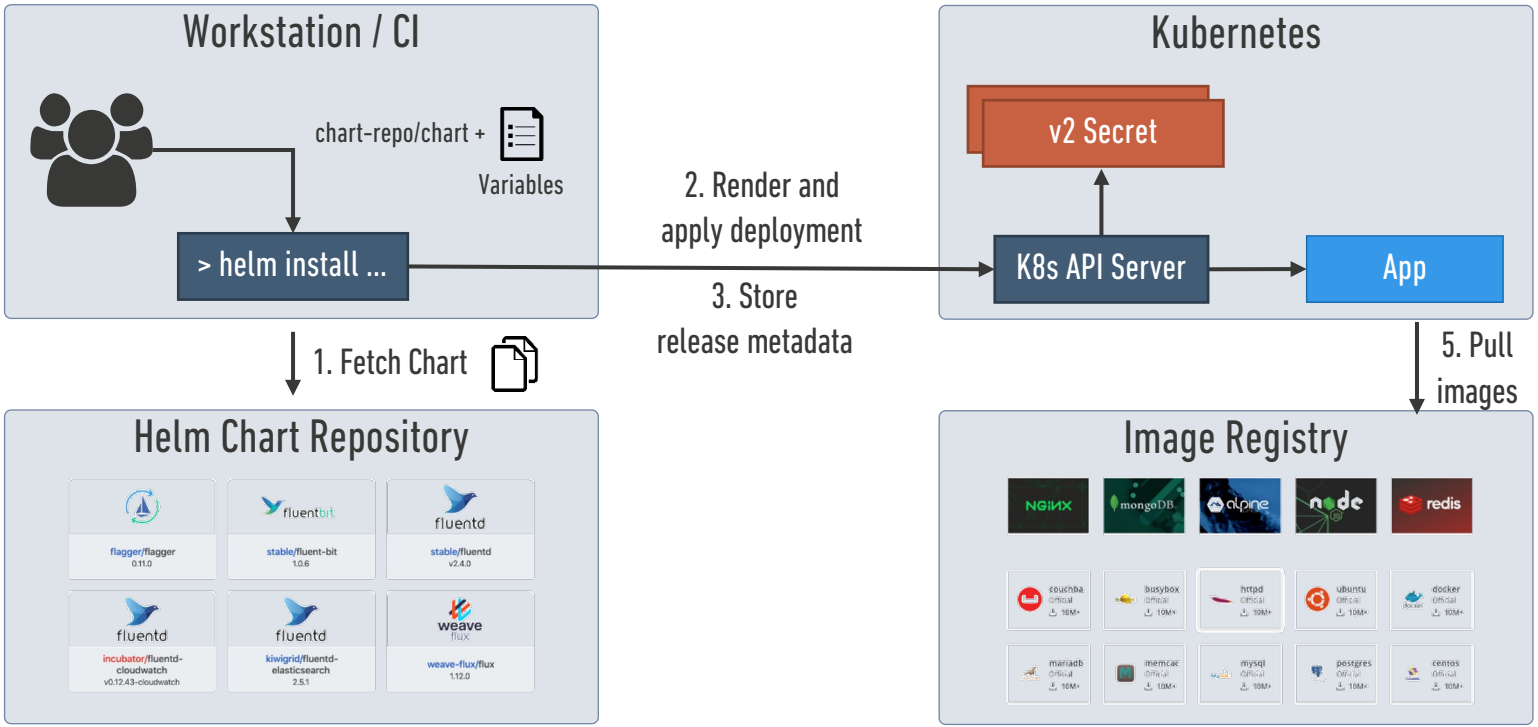
HELM

OVERVIEW

- Tool for managing Kubernetes applications (charts)
- Charts help to define, install, and upgrade Kubernetes applications
- Open-Source project developed by Microsoft, Google, Bitnami
- Templating based approach
- Graduated the Cloud-Native Computing Foundation (CNCF)

HELM

OVERVIEW



HELM CLIENT

- Command-line client for end users
- Local chart development
  - Create new charts from scratch
  - Package charts into chart archive (tgz) files
- Manage repositories
- Compile final deployment based on provided variables
- Manage release metadata

HELM CHART

STRUCTURE

A chart is organized as a collection of files inside of a directory:

Chart.yaml	A YAML file containing information about the chart	
LICENSE	A plain text file containing the license for the chart	OPTIONAL
README.md	A human-readable README file	OPTIONAL
values.yaml	The default configuration values for this chart	
values.schema.json	A JSON Schema for imposing a structure on the values.yaml file	OPTIONAL
charts/	A directory containing any charts upon which this chart depends.	
crds/	Custom Resource Definitions	
templates/	A directory of templates that, when combined with values, will generate valid Kubernetes manifest files.	
templates/NOTES.txt	A plain text file containing short usage notes	OPTIONAL

HELM CHART

CHART.YAML

```
apiVersion: The chart API version, "v1" or "v2" (required)
name: The name of the chart (required)
version: A SemVer 2 version (required)
kubeVersion: A SemVer range of compatible Kubernetes versions (optional)
description: A single-sentence description of this project (optional)
type: The type of the chart (optional)
keywords:
  - A list of keywords about this project (optional)
home: The URL of this project's home page (optional)
sources:
  - A list of URLs to source code for this project (optional)
dependencies: # A list of the chart requirements (optional)
...
maintainers: # (optional)
  - name: The maintainer's name (required for each maintainer)
    email: The maintainer's email (optional for each maintainer)
    url: A URL for the maintainer (optional for each maintainer)
icon: A URL to an SVG or PNG image to be used as an icon (optional).
appVersion: The version of the app that this contains (optional). This needn't be SemVer.
deprecated: Whether this chart is deprecated (optional, boolean)
annotations:
  example: A list of annotations keyed by name (optional).
```

https://semver.org/

HELM CHART

CHART VERSIONING

- Every chart must have a version number that follows the Semantic Versioning 2.0.0 schema
- Given a version number MAJOR.MINOR.PATCH(.LABEL), increment the:
  - MAJOR version when you make incompatible API changes,
  - MINOR version when you add functionality in a backwards-compatible manner, and
  - PATCH version when you make backwards-compatible bug fixes.
  - LABEL can be used for pre-release and build metadata
- Version number is used by Helm as release marker
- Packages in repositories are identified by name plus version
  - tomcat-sample-1.2.3.pre-release.tgz
  - CAUTION: Version in Chart.yaml must match version in package name!
- `appVersion` field in Chart.yaml refers to software version deployed by chart (e.g. nginx 1.15.12), and has no impact on chart version

HELM CHART

CHART LICENSE, README AND NOTES

- LICENSE
  - Plain text file containing the license for the chart
  - Can also contain license for application installed by chart
- README.md
  - Formatted in Markdown
  - Description of the application or service the chart provides
  - Any prerequisites or requirements to run the chart
  - Descriptions of options in values.yaml and default values
  - Any other information that may be relevant to the installation or configuration of the chart
- templates/NOTES.txt
  - Printed out after installation, and when viewing the status of a release
  - Can be used to display usage notes, next steps, or any other information relevant to a release of the chart
  - Is templated and therefor can display values evaluated at deployment time

HELM CHART

TEMPLATES & VALUES

- Helm Chart templates are written in the Go template language
- > 50 additional add-on template functions are available
- All template files are stored in a chart's `templates/` folder
- Values for the templates are supplied two ways
  1. A file called `values.yaml` inside of a chart
  2. Users of chart may supply a YAML file that contains values via command line
     ```
     $ helm install --values my-values.yaml ...
     ```
- Values from command-line override values from chart's `values.yaml`

HELM CHART

TEMPLATES & VALUES

- Template files follow the standard conventions for writing Go templates

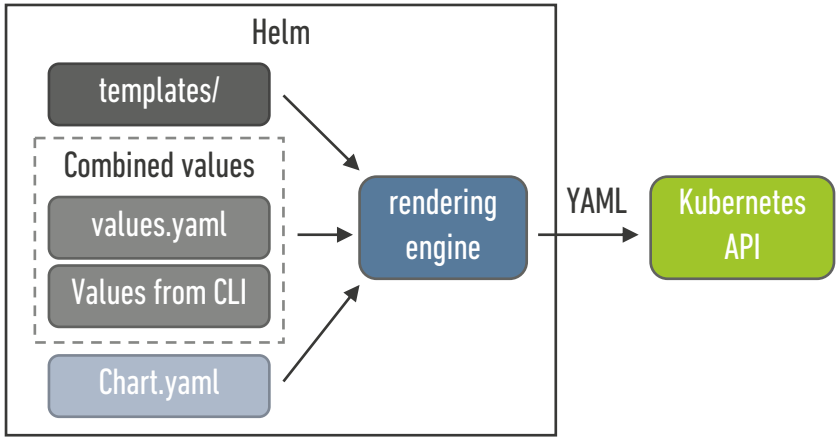
```yaml
# values.yaml
service:
  type: ClusterIP
  port: 80
```

```yaml
# templates/service.yaml
apiVersion: v1
kind: Service
metadata:
  name: {{ .Release.Name }}
  labels:
    app.kubernetes.io/instance: {{ .Release.Name }}
    app.kubernetes.io/managed-by: {{ .Release.Service }}
spec:
  type: {{ .Values.service.type }}
  ports:
    - port: {{ .Values.service.port }}
      targetPort: http
      protocol: TCP
      name: http
  selector:
    app.kubernetes.io/instance: {{ .Release.Name }}
```

HELM CHART

TEMPLATES & VALUES

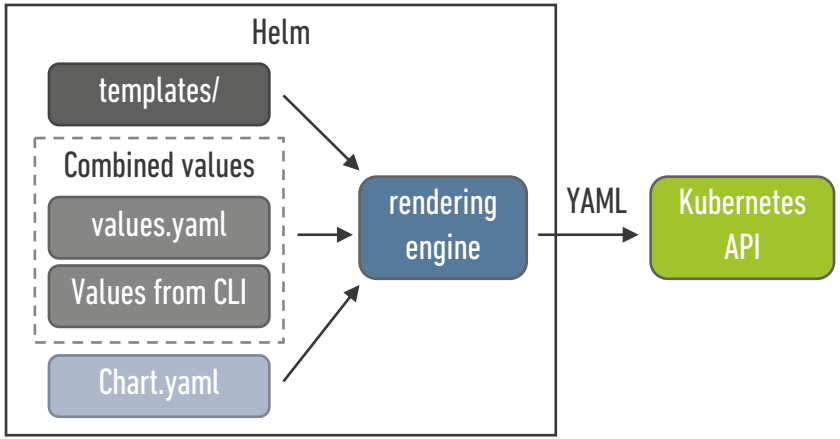
- The `templates/` directory is for template files
- Templates generate Kubernetes YAML-formatted manifest files
- Files starting with `_` will not be rendered to Kubernetes objects but can be used everywhere within other chart templates



HELM CHART

TEMPLATES & VALUES

- Different ways to install with custom values
  - Using custom values file
    `helm install -f myvalues.yaml …`
  - Setting individual values
    `helm install --set container.image=busybox:latest …`



HELM CHART

CRDS

- Kubernetes CRDs are treated as a special kind of object
- Installed before the rest of the chart
- CRD files cannot be templated!
- Limitations
  - CRDs are never reinstalled
  - CRDs are never installed on upgrade or rollback
  - CRDs are never deleted

HELM CHART REPOSITORY

- HTTP server that houses one or more packaged charts
- Has to be able to serve YAML files and tar files via GET requests
- Helm comes with built-in package server for developer testing
  `$ helm serve`
- Other serves have been tested
  - Google Cloud Storage with website mode enabled
  - S3 with website mode enabled
- Every repository requires a `index.yaml`
- Helm **does not** provide tools for uploading charts to remote repository servers

```
charts/
  |
  |- index.yaml
  |
  |- alpine-0.1.2.tgz
  |
  |- alpine-0.1.2.tgz.prov
```

HELM CHART REPOSITORY

- The `index.yaml` contains
  - List of all of the packages supplied by the repository
  - Metadata that allows retrieving and verifying packages
- The Helm project provides an open-source Helm repository server called ChartMuseum
  - Supports multiple cloud storage backends
  - `index.yaml` file will be generated dynamically
  - API for chart uploads

```yaml
apiVersion: v1
entries:
  alpine:
  - created: 2016-10-06T16:23:20.499543808-06:00
    description: Deploy a basic Alpine Linux pod
    digest: 515c58e5f79d8b2913a10cb400ebb6fa9c77fe813287afbacf1a0b897cd78727
    home: https://k8s.io/helm
    name: alpine
    sources:
    - https://github.com/helm/helm
    urls:
    - https://technosophos.github.io/tscharts/alpine-0.1.2.tgz
    version: 0.1.2
generated: 2016-10-06T16:23:20.499029981-06:00
```

HELM PROVENANCE AND INTEGRITY

- Helm has provenance tools which help chart users verify the integrity and origin of a package
- Integrity is established by comparing a chart to a provenance record
- Provenance records are stored in provenance files next to the chart archive
  - Contains a chart's YAML file
  - The SHA256 signature of the chart package (.tgz file)
  - The entire body is signed using the algorithm used by PGP
- Provenance files are generated at packaging time

```
$ helm package --sign ...
$ helm install --verify
```

- Helm uses GnuPG for key handling and signing

HELM

WORKING WITH CHARTS

WORKING WITH CHARTS

HELM COMMANDS

- Helm Client provides different commands for discovering charts
  - `repo` - Add, list, remove, update, and index chart repositories
  - `search` – Search for a keyword in charts
  - `show` – Prints the contents of the Chart.yaml file and the values.yaml file
  - `show chart` – Prints the contents of the Chart.yaml
  - `show readme` – Prints the contents of the README
  - `show values` - Prints the contents of the values.yaml
  - `pull` – Download a chart from a repository and (optionally) unpack it in local directory
  - `verify` - Verify that the given chart has a valid provenance file

WORKING WITH CHARTS

HELM COMMANDS

- From a chart, a release can be created and managed
  - `install` – Installs a chart archive
  - `test` - Runs tests defined by the chart for a release
  - `status` - Displays the status of the named release
  - `get` – Shows the details of a named release
  - `get manifest` – Fetches the generated Kubernetes manifests for a given release
  - `get notes` – Shows notes provided by the chart of a named release
  - `get values` – Downloads a values file for a given release
  - `upgrade` - Upgrades a release to a specified version of a chart and/or updates chart values
  - `history` - Prints historical revisions for a given release
  - `uninstall` - Given a release name, delete the release from Kubernetes

HANDS-ON

WORKING WITH CHARTS

WORKING WITH CHARTS

INSTALL CHARTMUSEUM

1. Search for chart “choerodon/chartmuseum” on artifacthub.io
2. Add repo to your helm client as described there

WORKING WITH CHARTS

INSTALL CHARTMUSEUM

4. Inspect the "choerodon/chartmuseum" chart and create a `chartmuseum-values.yaml` file (https://artifacthub.io/packages/helm/choerodon/chartmuseum)
   - Enable API
   - Service type LoadBalancer
   - Enable persistence:
     - ~~use local-nfs storageclass~~
     - 1GB storage
5. Deploy Chartmuseum via Helm using above `chartmuseum-values.yaml` file
   - Add repo
   - Name → chartmuseum
   - Namespace → helm-sample
   - Inspect chartmuseum service and open endpoint in browser
6. Add Chartmuseum to your Helm repos

```
$ helm repo add chartmuseum http://<EXTERNAL_IP>:8080/
```

```
#chartmuseum-values.yaml

image:
  repository: TODO
  tag: TODO
service:
  type: TODO
…
…
#TODO some other params
```

HELM

CHART CREATION

CHART CREATION

HELM COMMANDS

- Helm provides different commands for chart creation
  - `create` - Creates a chart directory along with the common files and directories used in a chart
  - `dependency` – Build, list and update chart dependencies found in the `charts/` folder
  - `lint` - Takes a path to a chart and runs a series of tests to verify that the chart is well-formed
  - `package` - Packages a chart into a versioned chart archive file
- Chart creation best-practices
  - Chart names should use lower case letters and numbers, and start with a letter
  - Variable names in `values.yaml` should begin with a lowercase letter, and words should be separated with camelCase
  - ...

  The list goes on → https://helm.sh/docs/chart_best_practices/

CHART CREATION

TEMPLATES

- Default content of `templates/`
  - `NOTES.txt` – Will be displayed to users when they run helm install
  - `deployment.yaml` – A basic manifest for creating a Kubernetes deployment
  - `service.yaml` – A basic manifest for creating a service for your deployment
  - `_helpers.tpl` – A place to put template helpers that can be re-use throughout the chart
- Within files inside the `templates/` folder, we can insert variables replaced at deploy time
  - `{{ .Release.Name }}`
  - `{{ .Values.image.repository }}`

CHART CREATION

TEMPLATES

- Helm provides build-in objects (https://helm.sh/docs/chart_template_guide/#built-in-objects)
  - `Release` - Describes the release itself
  - `Values` - Values passed into the template from the `values.yaml` file and from user-supplied files
  - `Chart` - The contents of the `Chart.yaml` file
  - `Files` - Provides access to all non-special files in a chart (no templates!)
  - `Capabilities` - Information about what capabilities the Kubernetes cluster supports
  - `Template` - Information about the current template that is being executed

CHART CREATION

TEMPLATES

- Objects can be processed using functions and pipelines (https://helm.sh/docs/chart_template_guide/#template-functions-and-pipelines)
  - `{{ quote .Values.favorite.food }}`
  - `{{ .Values.favorite.food | upper | quote }}`
  - `{{ .Values.favorite.drink | repeat 5 | quote }}`
- Helm has over 60 available functions
  - Go template language
  - Sprig template library
- Functions follow the syntax `functionName arg1 arg2 ...`
- Pipelines are a tool for chaining together a series of functions
- Pipelines invert order by passing along values to function via `|`

CHART CREATION

TEMPLATES

- Handling white-spaces in templates
  - `{{` and `}}` are removed, but remaining whitespace stay exactly as is
  - As YAML ascribes meaning to whitespace this becomes an issue
  - `{{-` indicates that whitespace should be chomped left
  - `-}}` indicates that whitespace should be chomped right
  - Newlines are whitespace!

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "tomcat-sample.fullname" . }}
data:
  color: {{ .Values.color | quote }}
  {{ if and .Values.color (eq .Values.color "blue") }}
  easter-egg: true
  {{ else }}
  easter-egg: false
  {{ end }}
```

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: harping-ocelot-tomcat-sample
data:
  color: "blue“

  easter-egg: true
```

HELM

CONTROL FLOW

CHART CREATION

TEMPLATES

- Control flow of the generation can be managed with following structures
  - `if` / `else` – Creates conditional blocks
  - `with` – Defines a scope
  - `range` – Iterate through a list of values

```
{{ if PIPELINE }}
# Do something
{{ else if OTHER PIPELINE }}
# Do something else
{{ else }}
# Default case
{{ end }}
```

```
{{ with PIPELINE }}
# restricted scope
{{ end }}
```

```
{{ range SLICE }}
# current value -> .
{{ end }}
```

CHART CREATION

TEMPLATES

- `if` / `else` can evaluate a simple value or an entire pipeline
- A pipeline is evaluated as `false` if the value is
  - Boolean false
  - Numeric zero
  - Empty string
  - Nil (empty or null)
  - Empty collection (`map`, `slice`, `tuple`, `dict`, `array`)

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "tomcat-sample.fullname" . }}
data:
  color: {{ .Values.color | quote }}
  easter-egg: {{ if and .Values.color (eq .Values.color "blue") }}true{{ else }}false{{ end }}
  sample.txt: |-
    This is a sample!
    And has multiple lines...
```

HELM

WITH & RANGE

CHART CREATION

TEMPLATES

- `with` controls variable scoping
- `.` is a reference to the current scope
  → `.Values` tells template to find Values object in current scope
- `with` allows you to set the current scope (`.`) to a particular object
- Inside of the restricted scope, other objects from the parent scope can not be accessed

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ include "tomcat-sample.fullname" . }}
data:
  {{- with .Values }}
  color: {{ .color | quote }}
    {{- if and .color (eq .color "blue") }}
  easter-egg: true
    {{- else }}
  easter-egg: false
    {{- end }}
  {{- end }}
```

CHART CREATION

TEMPLATES

- `range` operator is used to iterate through a collection
- Can handle simple lists and key/value objects like `map` or `dict`
- The collection can come from Values or can be quickly created inside of a template via `tuple`

```
colors: |-
  {{- range .Values.availableColors }}
  - {{ . | title | quote }}
  {{- end }}
```

```
colors: |-
  {{- range tuple “red" “green" “blue" }}
  - {{ . | title | quote }}
  {{- end }}
```

HELM

VARIABLES

CHART CREATION

TEMPLATES

- A variable is a named references to another objecs in the template
- It follows the form `$name`
- Variables can simplify code and allow a better use of `with` and `range`
- Variable `$` is always global and points to the root context

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  {{- $relname := .Release.Name -}}
  {{- with .Values.favorite }}
  drink: {{ .drink | default "tea" | quote }}
  food: {{ .food | upper | quote }}
  release: {{ $relname }}
  chartName: {{ $.Chart.Name }}
  {{- end }}
```

```
availableColors: |-
  {{- range $index, $color := .Values.availableColors }}
  {{ $index }}: {{ $color }}
  {{- end }}
```

HELM

NAMED TEMPLATES

CHART CREATION

TEMPLATES

- With “named templates”, parts of templates can be reused throughout the chart
- Named templates are defined inside a file and given a name
- Template names are global
  - If two templates have the same name, whichever one is loaded last will be the one used
  - This includes sub-charts

  → Prefix each templates with the name of the chart

CHART CREATION

TEMPLATES

- The `define` action creates a named template inside of a template file
- By convention, these templates are consolidated in a a partials file, usually `_helpers.tpl`
- `define` functions should have a simple documentation block (`{{/* ... */}}`) describing what they do

- The `template` includes the sub-template or partial
- The scope has to be passed into the template call

```
# _helpers.tpl
{{- define "tomcat-sample.labels" }}
  labels:
    generator: helm
    date: {{ now | htmlDate }}
    chart: {{ .Chart.Name }}
    version: {{ .Chart.Version }}
{{- end }}
```

```
# other template
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "tomcat-sample.labels" . }}
```

CHART CREATION

TEMPLATES

- When using `template` action, the data is simply inserted inline
- It can not be processed via other functions

```
# _helpers.tpl
{{- define "tomcat-sample.labels" }}
labels:
  generator: helm
  date: {{ now | htmlDate }}
  chart: {{ .Chart.Name }}
  version: {{ .Chart.Version }}
{{- end }}
```

+

```
# other template
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- template "tomcat-sample.labels" . }}
```

=

```
# Result
apiVersion: v1
kind: ConfigMap
metadata:
  name: abc-def-configmap
labels:
  generator: helm
  date: 01-01-2019
  chart: tomcat-sample
  version: 0.1.0
```

CHART CREATION

TEMPLATES

- Helm provides an alternative to `template`
  → `include`
- `include` will import the contents of a template into the present pipeline
- Now it can be processed using any given function

```
# _helpers.tpl
{{- define "tomcat-sample.labels" }}
labels:
  generator: helm
  date: {{ now | htmlDate }}
  chart: {{ .Chart.Name }}
  version: {{ .Chart.Version }}
{{- end }}
```

+

```
# other template
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
  {{- include "tomcat-sample.labels" . | nindent 2 }}
```

=

```
# Result
apiVersion: v1
kind: ConfigMap
metadata:
  name: abc-def-configmap
  labels:
    generator: helm
    date: 01-01-2019
    chart: tomcat-sample
    version: 0.1.0
```

HELM

FILE ACCESS

CHART CREATION

TEMPLATES

- Helm provides access to files through the `.Files` object
- Some files cannot be accessed through the `.Files` object
  - Files in `templates/` cannot be accessed.
  - Files excluded using `.helmignore` cannot be accessed.
- CAUTION: Charts must be smaller than 1M because of the storage limitations of Kubernetes objects

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: {{ .Release.Name }}-configmap
data:
  sample.txt: |-
    {{ .Files.Get "sample.txt" }}
```

CHART CREATION

TEMPLATES

- Helm imports many of the functions from Go's path package
- The imported functions are:
  - base
  - dir
  - ext
  - isAbs
  - clean
- Files.Glob(pattern string) method assists in extracting certain files with all the flexibility of glob patterns

```
{{ $root := . }}
{{ range $path, $bytes := .Files.Glob "**.yaml" }}
{{ $path }}: |-
{{ $root.Files.Get $path }}
{{ end }}
```

CHART CREATION

TEMPLATES

- It is very common to want to place file content into both ConfigMaps and Secrets
- The content of a file can be processed by functions for example for base-64 encoding
- Each line of a file can be accessed via the `Lines` method

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: conf
data:
  {{- (.Files.Glob "foo/*").AsConfig | nindent 2 }}
---
apiVersion: v1
kind: Secret
metadata:
  name: very-secret
type: Opaque
data:
  {{- (.Files.Glob "bar/*").AsSecrets | nindent 2 }}
```

```
apiVersion: v1
kind: Secret
metadata:
  name: {{ .Release.Name }}-secret
type: Opaque
data:
  token: |-
    {{ .Files.Get "config1.toml" | b64enc }}
```

```
data:
  some-file.txt: {{ range .Files.Lines "foo/bar.txt" }}
    {{ . }}{{ end }}
```

HELM

PUBLISHING CHARTS

CHART CREATION

SETUP CHART

1. Package chart
   ```
   $ helm package
   ```
2. Upload chart to Chartmuseum
   ```
   $ curl --data-binary "@tomcat-sample-0.1.0.tgz" http://<CLUSTER-IP>:8080/api/charts
   ```
3. Update local index of Chartmuseum repo
4. Create new release from chart chartmuseum/tomcat-sample in helm-sample namespace