Copyright Information

Published in the United States by Alta3 Research, Inc.

4 Bonnywick Drive
Harrisburg, PA 17111
Phone: *717-56 6 -4428*
Web: *https://alta3.com*
Email: *info@ alta3.com*

Welcome!

Thank you for selecting Alta3 Research as your training provider! Since 1997, Alta3 Research has been empowering organizations and individuals using a no-nonsense approach to IT and DevOps training. We specialize in technologies such as Python, Ansible, 5G, Software Defined Networking, Microservices, Kubernetes, GoLang, Jenkins, GitHub, GitLab and more.

Alta3 Research excels at taking the advanced technologies required for IT and DevOps professionals, breaking them down and building proven training courses with over 95% student satisfaction. Training is available at the customer's site or virtually through instructor-led webinars. For more information on additional courses to help you achieve your career goals, please visit us at https://alta3.com

PDF Content Expiration

- **PDF created: 2023-02-14**

- **Refresh your content at: https://alta3.com**

Subscribe to us on these social media platforms for updates and free training!

- **YouTube:** https://youtube.com/alta3research
- **Twitter:** @alta3research
- **Facebook:** https://facebook.com/alta3research
- **LinkedIn:** https://www.linkedin.com/company/alta3-research-inc

Table of Contents

Kubernetes Bootcamp

1. Welcome to the Alta3 Research Lab Environment

Lab Objective

These labs will guide you through **hands-on, step-by-step exercises**, with *detailed explanations for each step*.

The virtual environment you will use to perform these labs is available to you **24/7** throughout the duration of your course. *We do not turn them off in between sessions*.

Getting Started

Let's make sure you are ready to start running through these labs.

Even so, you may have to use a new key-combination to use the copy-paste functionality. Let's practice it now.

If you click on the little clipboard icon in a lab, it will copy whatever text or code block is next to it, and will be available to paste into the virtual machine.

```
student@bchd:~$ # copy me!
```

After clicking on the , you will then be able to paste the command or code block wherever you would like.

Typically, you will want to paste into the Virtual Machine's terminal.

Each browser is different, but there are several ways to do this:

1. In the terminal, right click, then select `paste` (or) `paste as plain-text` (depending on the browser)
2. `Shift` `Insert`
3. `Ctrl` `Shift` `v`

Go ahead and try to paste the text of `# copy me!` into your terminal now.

Fundamental Commands

Not every person in this course has experience using a terminal to interact with a computer. This section is for those who may not have previous experience using a Linux terminal.

First of all, take a look at your command prompt to try to understand what it can teach us. It should look like:

```
student@bchd:~$
```

- **student** is the name of the user
- **bchd** is the hostname of the machine
- **~** (to the right of the colon) shows us the present working directory. Specifically, **~** refers to this user's home directory of `/home/student`.
- **$** shows us that this is a typical user (vs. **#** would indicate that it is the **root** user)

Now let's run some fundamental commands to get to know our environment a little bit better. Remember, we are starting in our `/home/student` directory for this.

1. **pwd** - [present working directory] shows you what directory you are in.

   ```
   student@bchd:~$ pwd

   /home/student
   ```

2. **ls** - list out the contents of the current directory.

   ```
   student@bchd:~$ ls

   static
   ```

3. **cd** - [change directory] - allows you to move to a different directory. Here we can move to the static directory.

   ```
   student@bchd:~$ cd static
   ```

4. **mkdir** - [make directory] - allows you to make a new directory. Let's make one called **training**.

```
student@bchd:~/static$ mkdir training
```

Also, let's make sure we can see that we made this **training** directory.

```
student@bchd:~/static$ ls

training
```

And let's move into the **training** directory.

```
student@bchd:~/static$ cd training
```

5. **touch** - makes a blank file.

```
student@bchd:~/static/training$ touch example_01.txt
```

Verify that the file named **example_01.txt** is there now.

```
student@bchd:~/static/training$ ls

example_01.txt
```

6. **echo** - returns text to the standard output.

```
student@bchd:~/static/training$ echo Alta3 Research Training rocks!

Alta3 Research Training rocks!
```

The > character allows us to redirect standard output to a different place, normally a file.

```
student@bchd:~/static/training$ echo Alta3 Research has AWESOME labs! > myfile.txt
```

The >> characters allow us to redirect standard output and append it to the end of a file.

```
student@bchd:~/static/training$ echo Alta3 Research has AMAZING labs! >> myfile.txt
```

7. **cat** - [concatenate] prints file(s) to standard output.

```
student@bchd:~/static/training$ cat myfile.txt

Alta3 Research has AWESOME labs!
Alta3 Research has AMAZING labs!
```

8. **mv** - [move] allows us to move a file or directory (often used to rename files).

```
student@bchd:~/static/training$ mv myfile.txt ego_fuel.txt
```

Verify that the file has moved.

```
student@bchd:~/static/training$ ls

ego_fuel.txt example_01.txt
```

Make sure that the contents of the file have not changed.

```
student@bchd:~/static/training$ cat ego_fuel.txt

Alta3 Research has AWESOME labs!
Alta3 Research has AMAZING labs!
```

9. **history** - shows all of the commands performed in this shell session.

```
student@bchd:~/static/training$ history
```

```
# copy me
pwd
ls
cd static
mkdir training
ls
cd training
touch example_01.txt
ls
echo Alta3 Research Training rocks!
echo Alta3 Research has AWESOME labs! > myfile.txt
echo Alta3 Research has AMAZING labs! >> myfile.txt
cat myfile.txt
mv myfile.txt ego_fuel.txt
ls
cat ego_fuel.txt
history
```

10. **man** - [manual] show the manual pages (aka documentation) for a given command. This is helpful for understanding any commands you may not feel comfortable with.

    `student@bchd:~/static/training$ man ls`

    To quit out of this view, press the keyboard key **q**

Excellent work! Now, let's move back to the home directory before you start working on the next lab.

`student@bchd:~/static/training$ cd ~`

Your prompt should now be back to: `student@bchd:~$`

If you have any questions throughout the course, please feel free to reach out to:

- Your Instructor
- Live Help via the Alta3 Research Discord Channel
- Email the Alta3 Research's Support Team support@alta3.com

Helpful Resources

Alta3 Research Instruction & Training

Alta3 Research Posters & Cheat Sheets

Alta3 Research YouTube

Common Questions and Solutions

- **My screen has "dots" around it :(**
  - *If you have more than one screen open, the "smallest" resolution wins. Therefore, the solution is to close the second tab you have open.*
- **I typed exit too many times and my CLI session closed!**
  - *Press the Refresh icon on your browser to refresh your session and create a new tmux session*
- **How do I get my content out of the tmux (CLI) environment?**
  - *The "best" way is probably by using git and GitHub. Alternatively, if you move content into the `~/static/` folder, you may access it by changing the "source" of your right-most pane in the split-screen session. To change the source, click on the icon that looks like three sheets of paper in the upper-right hand corner.*
- **Will the lab environments "shut off" this week?**
  - *No. The your lab environment is on 24x7, until the termination date.*
- **Can I close the tab to live.alta3.com**
  - *Yes! Unless you clear your internet cache, you should just be able to revisit your custom course link to regain access. If you are asked to log-in again, simply use the same email address and handle to regain access.*

2. Kubernetes Bootcamp

- **3 days**
- **Lecture & Labs**

Course Overview

This class prepares students for the Certified Kubernetes Application Developer (CKAD) exam. Kubernetes is a Cloud Orchestration Platform providing reliability, replication, and stability while maximizing resource utilization for applications and services. By the conclusion of this hands-on training you will go back to work with all necessary commands and practical skills to empower your team to succeed, as well as gain knowledge of important concepts like Kubernetes architecture and container orchestration. We prioritize covering all objectives and concepts necessary for passing the Certified Kubernetes Application Developer (CKAD) exam. You will build, command, and configure a high availability Kubernetes environment capable of demonstrating all "K8s" features discussed and demonstrated in this course. Your three days of intensive, hands-on training will conclude with a mock CKAD exam that matches the real thing.

What You'll Learn

All topics required by the CKAD exam, including: - Deploy applications to a Kubernetes cluster - Use Kubernetes primitives to implement common deployment strategies (e.g. blue/green or canary) - Define, build and modify container images - Implement probes and health checks - Understand multi-container Pod design patterns (e.g. sidecar, init and others) - Understand ConfigMaps - Create & consume Secrets - Troubleshooting and debugging tools - Provide and troubleshoot access to applications via services - Use Ingress rules to expose applications

Course Outline:

1. **From Containers to Kubernetes**
    - Kubernetes Architecture
    - Components of a K8s Cluster
    - Master Services
    - Node Services
    - Containers
    - Why Do We Need Kubernetes?
    - Define, Build and Modify Container Images (Docker)
2. **Pod Basics**
    - Pods
    - K8s Services
    - Creating a K8s Cluster
3. **Basics - kubectl Commands**
    - Kubernetes Resources
    - Kubernetes Namespace
    - Kubernetes Contexts
4. **Pods**
    - Namespaces and Fundamental Kubectl Commands
    - What is a Pod?
    - YAML Essentials
    - Pod Manifests
    - Create, Configure, Delete Pods
    - Understand API versioning and deprecations
5. **Container Health, Security, and Observability**
    - Accessing containers with Port-Forward
    - How to Access Running Pods with Kubectl exec
    - Understanding Container and Pod Lifecycles
    - Maintaining Container Health with Probes
    - Pod SecurityContexts
6. **Resource Management**
    - Namespace Resource Control

- Understanding Resource Requirements, Limits, and Quotas
- Monitoring Applications with Kubectl Top
- Authentication to the API
- Authorization with the API
- Admission Controllers the API uses

7. **Logging**
   - Utilize Container Logs
   - Advanced Logging Techniques (Log aggregation)
8. **Ephemeral Storage**
   - Creating Ephemeral Storage for Logging (Fluentd)
   - Consistent Configuration with ConfigMaps
   - Create and Consume Secrets
9. **Persistent Storage**
   - PersistentVolumes
   - PersistentVolumeClaims
   - StorageClasses
   - Configuring Persistent Volumes for Storage
10. **Multi-Container Pod Design**
    - Why Use Multi-Container Pods?
    - Init Containers
11. **Deployments**
    - Understanding Labels and Selectors
    - Annotations
    - ReplicaSets
    - Why Use Deployments instead of ReplicaSets?
    - Advantages of Deployments
    - Writing a Deployment Management
    - Version Controls with Deployments
    - Rolling Updates and Rollbacks with Deployments
    - Blue/Green and Canary Deployments
    - Horizontal Scaling with Deployments
12. **Jobs and Cronjobs**
    - Understand Jobs and CronJobs
13. **NetworkPolicy**
    - Controlling Connectivity with NetworkPolicies
14. **Services and Ingress**
    - Kubernetes Services
    - ClusterIP
    - NodePort
    - LoadBalancer
    - Provide Access to Applications via Services
    - Ingress Controllers
    - Use Ingress Rules to Expose Applications
15. **The Helm Package Manager**
    - Why Do We Need Helm?
    - Helm Charts
    - Helm Repositories
    - Use the Helm Package Manager to Extend Kubernetes
16. **Extending Kubernetes**
    - Understanding Custom Resource Definitions
    - Creating and Using Custom Resource Definitions

17. **CKAD**
    - Introduction to CNCF
    - CKAD Objectives
    - The CKAD Exam
    - Tips to Pass the CKAD Exam

Hands on Labs:

1. Define, build and modify container images
2. Deploy Kubernetes using Ansible
3. Isolating Resources with Kubernetes Namespaces
4. Create and Configure Basic Pods
5. Debugging via kubectl port-forward
6. Performing Commands inside a Pod
7. Implement probes and health checks
8. Understanding Security Contexts
9. Understanding and defining resource requirements, limits and quotas
10. Monitoring Applications with Kubectl Top
11. Authentication, Authorization, and Admission
12. Utilize Container Logs
13. Creating Ephemeral Storage for Fluentd Logging Sidecar
14. Consistent Configuration with ConfigMaps
15. Create and Consume Secrets
16. Using PersistentVolumeClaims for Storage
17. Understand the Init container multi-container Pod design pattern
18. Understanding Labels and Selectors
19. Writing a Deployment Manifest
20. Perform rolling updates and rollbacks with Deployments
21. Advanced Deployment Strategies
22. Horizontal Scaling with kubectl scale
23. Understand Jobs and CronJobs
24. Deploy a NetworkPolicy
25. Provide and troubleshoot access to applications via services
26. Use Ingress rules to expose applications
27. Use the Helm package manager to deploy existing packages
28. Custom Resource Definitions (CRDs)

Bonus Labs

1. Isolating Resources with Kubernetes Namespaces
2. Horizontal Pod Autoscaling
3. Best Practices for Container Customization
4. Understand the Sidecar Multi-Container Pod Design Pattern
5. Dynamically Provision PersistentVolumes with NFS
6. Tainted Nodes and Tolerations
7. Calicoctl
8. Deploy a Kubernetes Cluster using Kubeadm
9. Create ServiceAccounts for use with the Kubernetes Dashboard
10. Sourcing Secrets from HashiCorp Vault
11. VNC Desktop
12. Saving Your Progress With GitHub
13. CKAD Practice Drill
14. CKAD Exam Bookmarks
15. Cluster Access with Kubernetes Context
16. Listing Resources with kubectl get
17. Examining Resources with kubectl describe
18. Imperative vs. Declarative Resource Creation
19. Insert an Annotation
20. Create and Configure a ReplicaSet
21. Create a Cluster Docker Registry
22. Setting up a single tier service mesh
23. Troubleshooting
24. A Completed Project

25. Patching
26. Inspect Container Logging
27. Advanced Logging Techniques

Prerequisites (not mandatory)

- Basic Linux skills are helpful
- Familiarity with a text editor like vi, vim, or nano is helpful

Who Should Attend?

- Anyone who plans to work with Kubernetes at any level or tier of involvement
- Any company or individual who wants to advance their knowledge of the cloud environment
- Application Developers
- Operations Developers
- IT Directors/Managers

Follow on Courses:

- CKA
- Developing Microservices

3. Using Vi and Vim

Throughout the course, you'll find our documentation suggests using the vim text editor. Vim is an improved version of vi, so if you know vi, you'll just be refreshing some basic skills in this lab.

Vim is the editor of choice for many developers and power users. It's a "modal" text editor based on the vi editor written by Bill Joy in the 1970s for a version of UNIX. It inherits the key bindings of vi, but also adds a great deal of functionality that is missing from the original vi.

In most text editors, the alphanumeric keys are only used to input those characters unless they're modified by a control key. In vim, the mode that the editor is in determines whether the alphanumeric keys will input those characters or move the cursor through the document. This is what is meant by 'modal.' When you first enter vim, you enter in the command mode.

Procedure - Using Vim

1. Review ***(read-only)*** the following **vim** commands:
    - **To start editing changes by entering the --INSERT-- mode:**
        - press `i`
    - **To stop editing and return to command mode:**
        - press `ESC`
    - **To save and quit:**
        - press `SHIFT` + **`:`** *press SHIFT and the COLON keys at the same time*
        - type `wq` (write out and quit)
        - press `ENTER` to confirm
    - **To quit without saving:**
        - press `SHIFT` + **`:`** *press SHIFT and the COLON keys at the same time*
        - type `q!` (quit and ignore all changes)
        - press `ENTER` to confirm
2. Move to the student home directory.

```
student@bchd:~$ cd
```

3. Now create a text file within the vim environment.

```
student@bchd:~$ vim zork.test
```

4. Vim is entered in command mode. To write text, you'll need to change to **--INSERT--** mode. To begin writing text, press:
    - `i`
5. Notice in the bottom left corner of the screen it now says **--INSERT--**
6. Type a few sentences. Be sure to include some carriage returns, like the following:

```
West of House
You are standing in an open field west of a white house, with a boarded front door.
There is a small mailbox here.
```

7. Okay, great! Now leave **--INSERT--** mode, and return to command mode, by pressing the escape key.
    - `Esc`
8. Notice that **--INSERT--** no longer is at the bottom left of the screen. Generally, pressing the Escape key will always return you to the command mode.
9. Use the directional arrow keys on the keyboard to move the cursor around the screen.
10. Perform the following to save changes, and return to the command line.
    - press `SHIFT` + **`:`**

press SHIFT and then the COLON key at the same time

- type `wq` (write out and quit)
- press `ENTER` to confirm

11. Confirm the file saved correctly by printing its contents. We can use the **cat** command to catenate, or read data from files and print their contents to the screen.

```
student@bchd:~$ cat zork.test
```

12. Edit the file zork.test again.

```
student@bchd:~$ vim zork.test
```

13. Remember, you enter vim in command mode. Take advantage of that and press the following capital letter to jump to the end of the file:

    - press `SHIFT` + `g`

press SHIFT and the g keys at the same time

14. Press the following capital letter to begin appending at the end of the line (enter --INSERT-- mode at the end).

    - press `SHIFT` + `a`

press SHIFT and the a keys at the same time

15. You'll notice it says **--INSERT--** at the bottom left of your screen again. Once again you can type normally. Add additional text, such as the following:

```
West of House
You are standing in an open field west of a white house, with a boarded front door.
There is a small mailbox here.
Open mailbox
Opening the small mailbox reveals a leaflet.
Read leaflet
(leaflet taken)
"WELCOME TO ALTA3 LABS!"
```

16. Okay, great! Now leave **--INSERT--** mode, and return to command mode by pressing the escape key.

    - `Esc`

17. Perform the following to return to the command line **without** saving any changes.

    - press `SHIFT` + `:`

press SHIFT and the COLON keys at the same time

- type `q!` (quit without saving)
- press `ENTER` to confirm

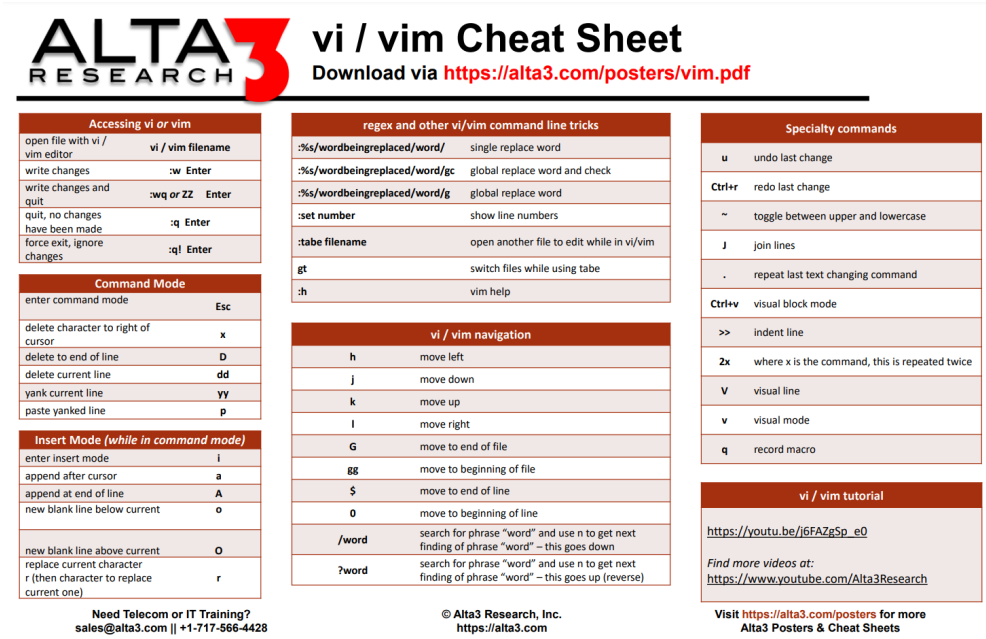
18. Confirm that none of the changes you just made were saved.

```
student@bchd:~$ cat zork.test
```

19. Remove the file.

```
student@bchd:~$ rm zork.test
```

20. Review a vim cheat sheet. These make very useful wall art and you may have seen one hanging in a colleague's workspace.



vi / vim Cheat Sheet

Download via https://alta3.com/posters/vim.pdf

Accessing vi *or* vim	
open file with vi / vim editor	vi / vim filename
write changes	:w Enter
write changes and quit	:wq *or* ZZ Enter
quit, no changes have been made	:q Enter
force exit, ignore changes	:q! Enter

Command Mode	
enter command mode	Esc
delete character to right of cursor	x
delete to end of line	D
delete current line	dd
yank current line	yy
paste yanked line	p

Insert Mode *(while in command mode)*	
enter insert mode	i
append after cursor	a
append at end of line	A
new blank line below current	o
new blank line above current	O
replace current character r (then character to replace current one)	r

Need Telecom or IT Training?
sales@alta3.com || +1-717-566-4428

regex and other vi/vim command line tricks	
:%s/wordbeingreplaced/word/	single replace word
:%s/wordbeingreplaced/word/gc	global replace word and check
:%s/wordbeingreplaced/word/g	global replace word
:set number	show line numbers
:tabe filename	open another file to edit while in vi/vim
gt	switch files while using tabe
:h	vim help

vi / vim navigation	
h	move left
j	move down
k	move up
l	move right
G	move to end of file
gg	move to beginning of file
$	move to end of line
0	move to beginning of line
/word	search for phrase "word" and use n to get next finding of phrase "word" – this goes down
?word	search for phrase "word" and use n to get next finding of phrase "word" – this goes up (reverse)

© Alta3 Research, Inc.
https://alta3.com

Specialty commands	
u	undo last change
Ctrl+r	redo last change
~	toggle between upper and lowercase
J	join lines
.	repeat last text changing command
Ctrl+v	visual block mode
>>	indent line
2x	where x is the command, this is repeated twice
V	visual line
v	visual mode
q	record macro

vi / vim tutorial
https://youtu.be/j6FAZgSp_e0
Find more videos at: https://www.youtube.com/Alta3Research

Visit https://alta3.com/posters for more Alta3 Posters & Cheat Sheets

Get all of Alta3's Cheat Sheets here! https://alta3.com/posters

21. Don't worry if you got stuck a few times, go back and try it again! Working within the vim environment is a basic Linux admin skill that is useful to everyone that expects to work at the Linux CLI.

4. Tmux

The program known as **tmux** is a very powerful tool for boosting the efficiency of using the command line. It allows you to *multiplex* the *terminal*, thus enabling you to effectively multiply the speed of your command line usage.

Learning Objective(s):

- Become familiar using **tmux** to massively increase your CLI efficiency

Procedure - Tmux

1. This lab is about learning to use tmux. You are already in a tmux session. If you weren't, you could apt-install **tmux**, then type the command `tmux` to get started. But the first thing we need to practice is a two-step process to enter into the command mode of **tmux**. Follow these steps:

    1. Hold the **Ctrl** key
    2. Tap the **B** key
    3. Take your hands off the keyboard (alias HOK)

    > Practice doing this seven times.

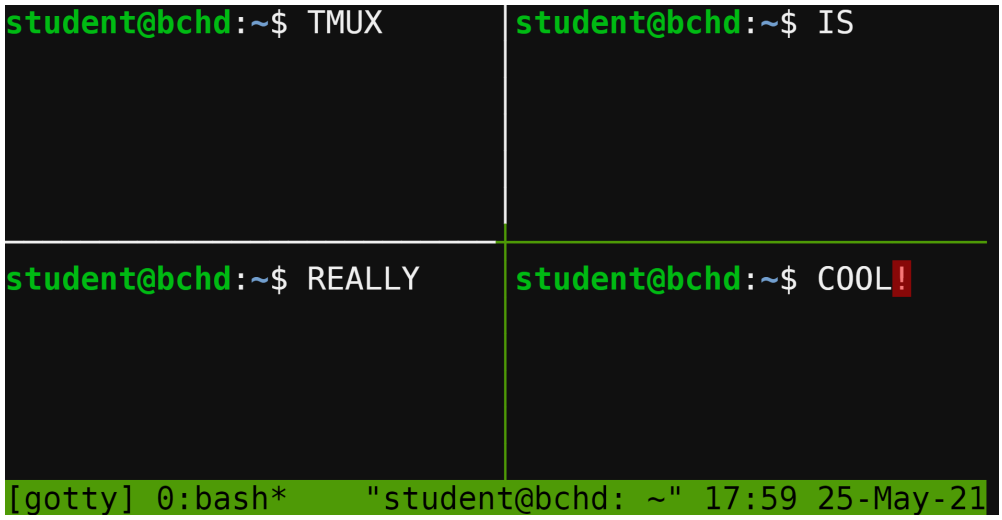
2. Now you are ready to learn some of the most essential **tmux** commands.

    For each of the following characters, enter the **tmux** command sequence (**Ctrl B**, HOK), then push the character. Note, not all of the characters are single-button presses. For example: `%` can be done by hitting the keyboard buttons `Shift` and `5`.

    - **%** - Vertical Split
    - **"** - Horizontal Split
    - **z** - Zoom/Unzoom
    - **←** - Move cursor leftwards one pane
    - **↑** - Move cursor upwards one pane
    - **→** - Move cursor rightwards one pane
    - **↓** - Move cursor downwards one pane
    - **[** - Scrollback (type `q` to exit)
    - **?** - Show the list of available tmux commands (type `q` to exit)

3. To exit out of a **tmux** pane, you can simply type the word `exit`, `logout`, or hit the key sequence **Ctrl D**.

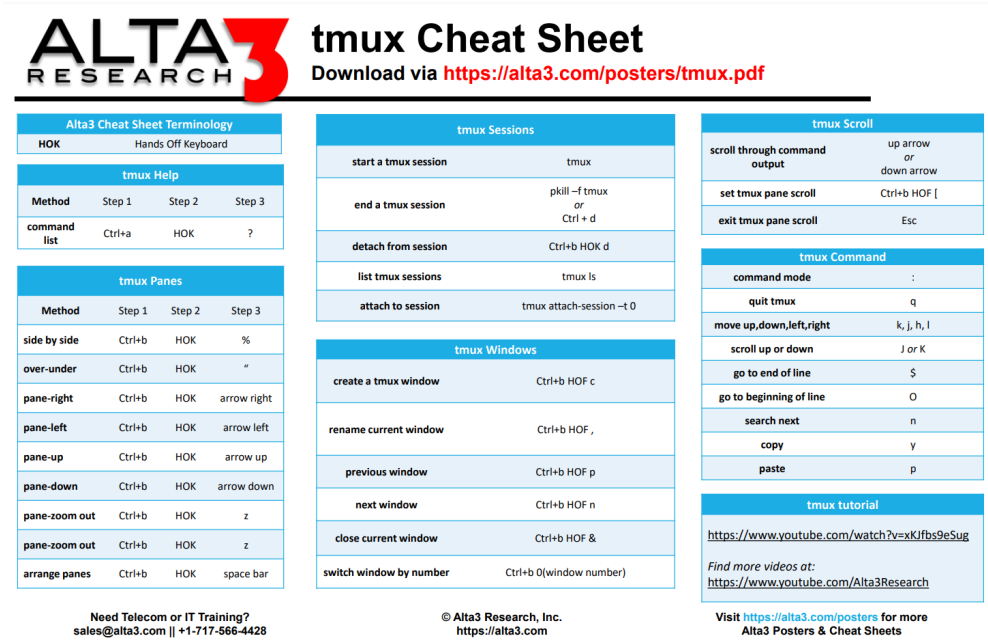
4. Challenge: Make your screen look like this:



    If you are struggling to do this, please re-watch the video or ask your instructor for help.

5. After completing this challenge, eliminate all but one of the panes before moving on to the next lab.

6. For more tmux commands, check out our tmux cheat sheet:



tmux Cheat Sheet

Download via https://alta3.com/posters/tmux.pdf

Alta3 Cheat Sheet Terminology	
HOK	Hands Off Keyboard

tmux Help			
Method	Step 1	Step 2	Step 3
command list	Ctrl+a	HOK	?

tmux Panes			
Method	Step 1	Step 2	Step 3
side by side	Ctrl+b	HOK	%
over-under	Ctrl+b	HOK	"
pane-right	Ctrl+b	HOK	arrow right
pane-left	Ctrl+b	HOK	arrow left
pane-up	Ctrl+b	HOK	arrow up
pane-down	Ctrl+b	HOK	arrow down
pane-zoom out	Ctrl+b	HOK	z
pane-zoom out	Ctrl+b	HOK	z
arrange panes	Ctrl+b	HOK	space bar

Need Telecom or IT Training?
sales@alta3.com || +1-717-566-4428

tmux Sessions	
start a tmux session	tmux
end a tmux session	pkill –f tmux *or* Ctrl + d
detach from session	Ctrl+b HOK d
list tmux sessions	tmux ls
attach to session	tmux attach-session –t 0

tmux Windows	
create a tmux window	Ctrl+b HOF c
rename current window	Ctrl+b HOF ,
previous window	Ctrl+b HOF p
next window	Ctrl+b HOF n
close current window	Ctrl+b HOF &
switch window by number	Ctrl+b 0(window number)

© Alta3 Research, Inc.
https://alta3.com

tmux Scroll	
scroll through command output	up arrow *or* down arrow
set tmux pane scroll	Ctrl+b HOF [
exit tmux pane scroll	Esc

tmux Command	
command mode	:
quit tmux	q
move up,down,left,right	k, j, h, l
scroll up or down	J *or* K
go to end of line	$
go to beginning of line	0
search next	n
copy	y
paste	p

tmux tutorial

https://www.youtube.com/watch?v=xKJfbs9eSug

Find more videos at:
https://www.youtube.com/Alta3Research

Visit https://alta3.com/posters for more Alta3 Posters & Cheat Sheets

Get all of Alta3's Cheat Sheets here! https://alta3.com/posters

5. K8S Architecture

Kubernetes - A Container Orchestrator

© Alta3 Research 1

Kubernetes
"Helmsman" in ancient Greek

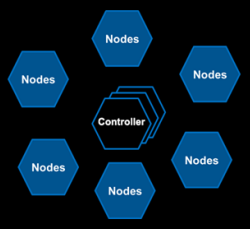
Architectural components:
1. External Load Balancer
2. Cluster
   - Control Plane
   - Worker Nodes
3. Pods
   - Containers
   - Registry
   - Pod Manifest (YAML)

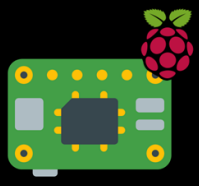
Worker Nodes

© Alta3 Research 2

A Worker node is a “PHYSICAL” machine in Kubernetes

- Previously known as a *minion*
- May be:
  - VM
  - Physical machine
- Provides the physical resources to run Pods
- Services on a node include
  - Container Runtime Environment
  - Kubelet
  - Kube-proxy

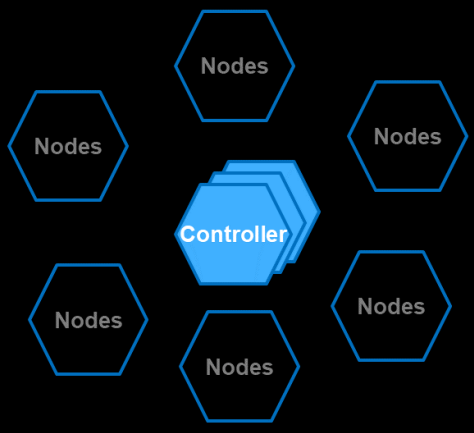




Control Plane

© Alta3 Research 3

K8’s Brain
- Previously known as Master
- Usually, three of them
- Manages nodes in its Kubernetes cluster
- Schedules Pods to run on those nodes
- Contains control plane components:
  - API server
  - Controller manager server
  - etcd



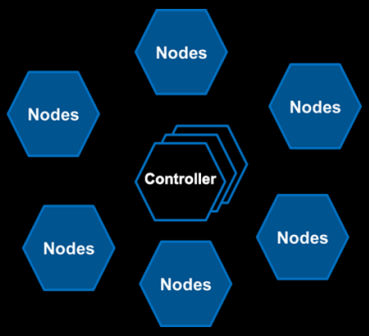
Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

Cluster

 4

A pool of nodes

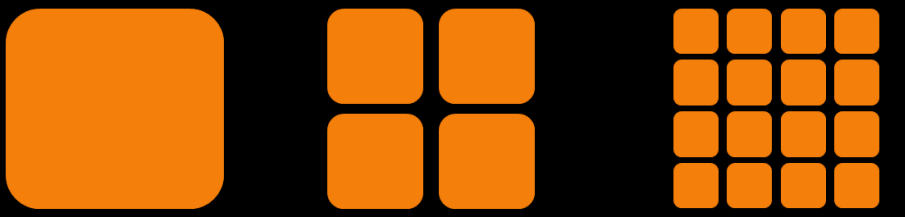
- Work is intelligently distributed across all the nodes
- The application programmer has no notion of the actual node running the code



It is possible to have multiple clusters

Microservices vs Monolithic

 5



Monolithic SOA Microservices

Monolithic

 6

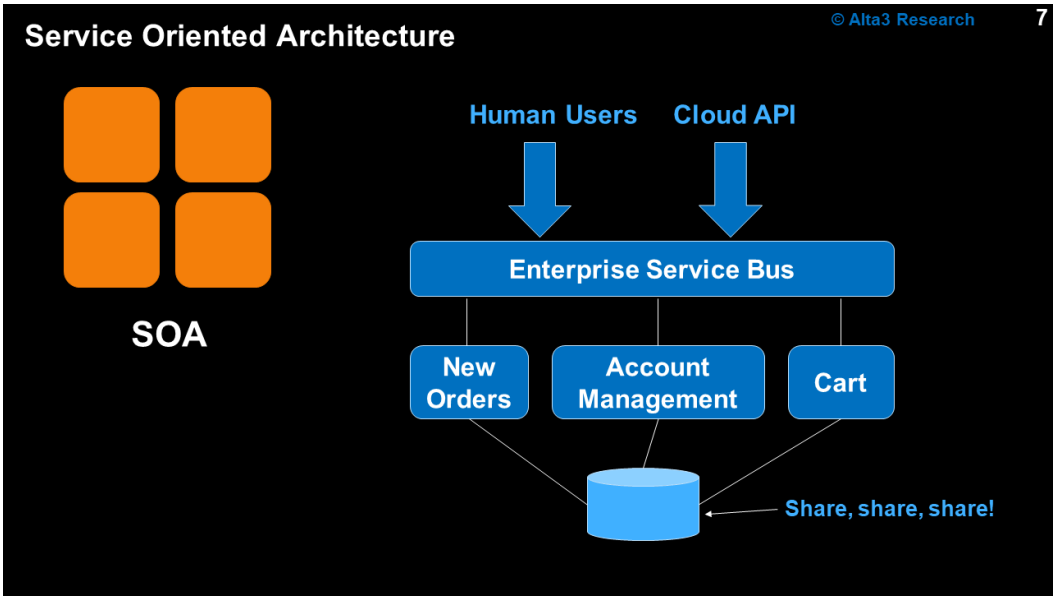
Monolithic

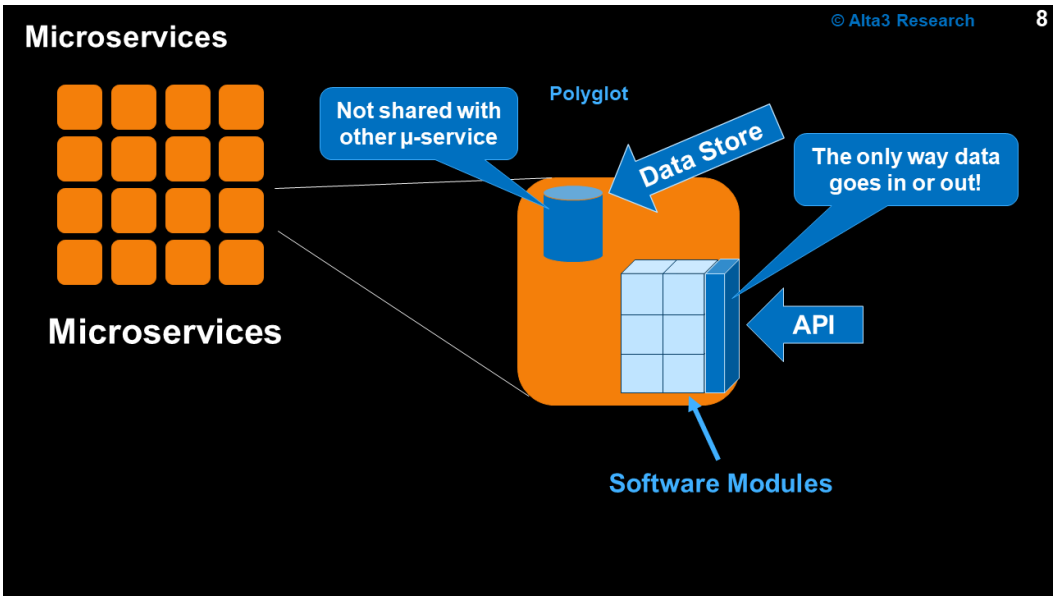
Built as a single unit with three parts:

1. Database (consisting of many tables)
2. Client-side user interface (HTML and/or JS)
3. Server-side application.

- A single logical executable.
- Alterations require a developer to build and deploy an updated version of these parts.

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com


Service Oriented Architecture
© Alta3 Research
7
SOA
Human Users
Cloud API
Enterprise Service Bus
New Orders
Account Management
Cart
Share, share, share!


Microservices
© Alta3 Research
8
Microservices
Not shared with other µ-service
Polyglot
Data Store
The only way data goes in or out!
API
Software Modules

6. Creating a Docker Image

Lab Objective

- Review or introduce Docker and Docker containers
- Understand the basics of a Container Runtime Environment (CRE)

Images are the basis for containers. In this lab, we use some existing code as the basis for an image. Then, we turn it into a Docker Container.

What is an image? An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image with some additional customization. For example, you may build an image which is based on the Ubuntu image but installs the Apache web server and your application as well as the configuration details needed to make your application run.

You might create your own images or you might only use those created by others and published in a registry. To build your own image, you create a Dockerfile with a simple syntax for defining the steps needed to create the image and run it. Each instruction in a Dockerfile creates a layer in the image. When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast when compared to other virtualization technologies. For simplicity, you can think of an image as akin to a git repository - images can be committed with changes and have multiple versions. If you don't provide a specific version number, the client defaults to **latest**.

Procedure

1. First, cd to your home directory.

```
student@bchd:~$ cd ~
```

2. Double check to make sure you have access to the docker group attached to it. If the first command does not show the word "docker" in the response, run the second command to add *docker* to the list of the student groups.

```
student@bchd:~$ groups student

student : student

student@bchd:~$ sudo usermod -aG docker $USER
```

3. In the next step, username ***student*** is passed with ***groups*** command and the output shows what groups ***student*** belongs to, separated by a colon.

```
student@bchd:~$ groups student

student : student docker
```

> Normally, you would reboot after adding docker to make these changes stick. However, you don't need to reboot in this lab environment. It is constructed this way to save you wait time.

4. Execute a new shell as the same user, which should get the new groups assigned.

```
student@bchd:~$ sudo -i -u $USER
```

5. BEFORE WE GO ANY FARTHER, we are going to install GoLang. We want to show you how to create small, compact, minimal containers that load blazingly fast, which shortens testing and greatly reduces future security risks because we are about to create a container that only contains the executable code and nothing else. GoLang is perfect for this job, so let's get started.

6. Install the Go language.

```
student@bchd:~$ sudo apt install -y golang
```

7. Confirm the installation by checking for the go version.

```
student@bchd:~$ go version
```

8. Create a hello world example by first setting up a working directory.

```
student@bchd:~$ mkdir greeter
```

9. Change directory into your working directory.

```
student@bchd:~$ cd greeter
```

10. Create your hello world program. Cut and paste the ENTIRE code block below into your terminal. The 7-line code block runs as a single command. The **cat >>EOF** portion of the command will start reading at the second line of the code block, ingesting all the lines until encountering an **EOF** string, which stops cat from reading in more content. Then the **> hello.go** portion of the command saves the ingested content into a file called **hello.go**.

```
student@bchd:~/greeter$

cat <<EOF > hello.go
package main
import "fmt"
func main() {
    fmt.Println("Hello, World!")
}
EOF
```

11. Execute the Hello World program with the following command:

```
student@bchd:~/greeter$ go run hello.go

Hello, World!
```

12. Now create a Go module. This is a file that takes care of a lot of details so we do not.

```
student@bchd:~/greeter$ go mod init greeter

go: creating new go.mod: module greeter
go: to add module requirements and sums:
    go mod tidy
```

13. OK! Now we can compile our code with the following command:

```
student@bchd:~/greeter$ go build
```

14. Let's see what just happened by listing the greeter directory.

```
student@bchd:~/greeter$ ls

greeter hello.go go.mod
```

15. Now we can run the executable file as follows:

```
student@bchd:~/greeter$ ./greeter

Hello, World!
```

16. Now that we see how to compile go code, let's compile a web server Go program that will serve a static web page which we will use in this class.

17. cd back to the home directory

```
student@bchd:~/greeter$ cd ~
```

18. Clone the static web server which includes both the web server and the static web content.

```
student@bchd:~$ git clone https://gitlab.com/alta3/webby.git
```

19. Change directory into the cloned server code.

```
student@bchd:~$ cd ~/webby
```

20. Now we will use a WONDERFUL feature in Go that will select the right go version to compile webby. Let's take advantage of that feature by turning it on.

```
student@bchd:~/webby$ export GO111MODULE="on"
```

21. Now let's activate a Go feature that will track our code's dependencies (so that we don't have to). We'll call our go module the same name as our web server "webby" purely for clarity.

```
student@bchd:~/webby$ go mod init webby
```

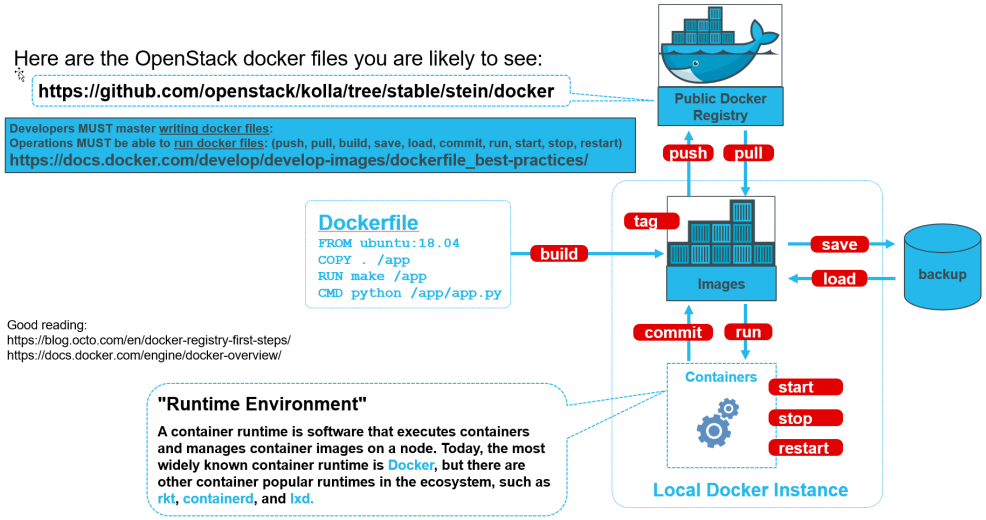
22. Next, let's install webby's dependencies. The go module we just created in the previous step will remember all of this.

```
student@bchd:~/webby$ go get -u github.com/stripe/stripe-go/v72

student@bchd:~/webby$ go get .
```

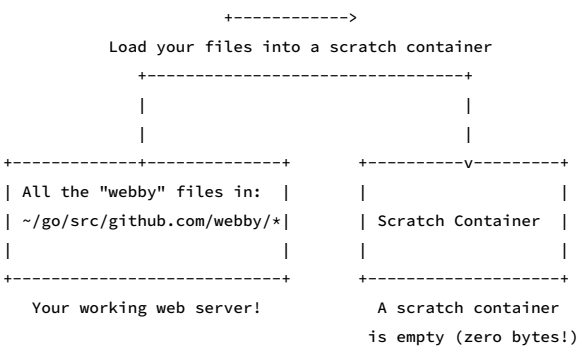
23. Here is the moment you have been waiting for! You will create a binary that will run autonomously inside a docker container. This tiny extra step is the difference between creating a 1GB container loaded with garbage and future security threats vs. a 100 MB sterile container that just works. This is why we are using go for this lab! Let's compile webby. It will take about a minute.

```
student@bchd:~/webby$ CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o webserver .
```

For your information, this is what the extra directives mean:
CGO_ENABLED=0 - Disable cgo to create a static binary
GOOS=linux - (GO Operating System) compiles the OS hooks INTO the binary
-a - flag means to rebuild all the packages we're using
-installsuffix cgo - Look for packages in $GOROOT/pkg/$GOOS_$GOARCH_cgo
-o webserver - Write compiled code to a file named `webserver`



24. Now that we have an image compiled, we need to load it into a totally EMPTY container. In Docker, an empty container is called a "scratch" container. Study this diagram:



25. Let's take a look at the scratch Dockerfile you just downloaded.

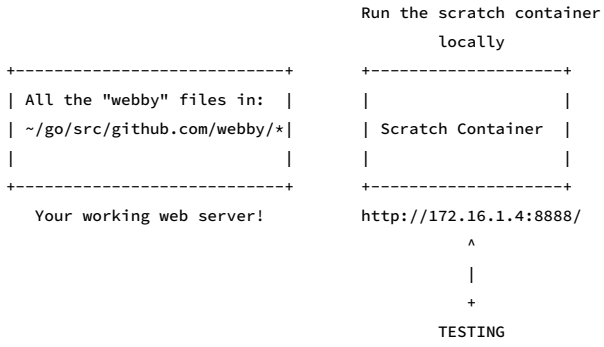
```
student@bchd:~/webby$ batcat Dockerfile
```

Click here to view the contents of **Dockerfile**

26. Use the Dockerfile you just created to create a lightweight image named **webby**:

```
student@bchd:~/webby$ sudo docker build -t webby -f Dockerfile .
```

```
Sending build context to Docker daemon    258MB
Step 1/5 : FROM scratch
 --->
Step 2/5 : ADD webserver /
 ---> ae076e08a34d
Step 3/5 : ADD deploy /deploy/
 ---> 01d62e9ad503
Step 4/5 : CMD ["/webserver"]
 ---> Running in 7a6080a0d6b2
Removing intermediate container 7a6080a0d6b2
 ---> 56e14f7fc894
Step 5/5 : EXPOSE 8888
 ---> Running in 339d6179f2e0
Removing intermediate container 339d6179f2e0
 ---> 03e6b74d7c12
Successfully built 03e6b74d7c12
Successfully tagged webby:latest
```



27. Run the container. Forward traffic from local port 2224 to remote port 8888. Then cd back to the home directory.

```
student@bchd:~/webby$ sudo docker run -p 2224:8888 --name mywebby -d webby

b63ccb7ee966
```

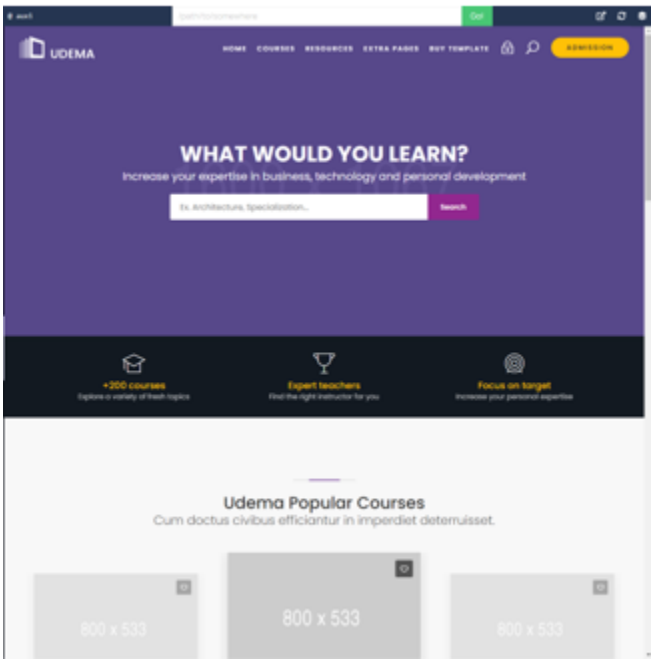
> Note: If you made a mistake in the file **Dockerfile**, you'll need to rebuild the image before attempting to run the container again.

```
student@bchd:~/webby$ cd ~
```

28. If you want to stick with your terminal environment for verification, run this command to see what is being served by our webby image.

```
student@bchd:~$ curl localhost:2224
```

29. If you would rather look at a more visual option, simply use your channel changer to select the appropriate **view** that is now being served. The channel changer can be found in the upper right hand corner, and looks like 3 sheets of paper. The **view** that you want is **aux1**. The **aux1** channel displays anything that is served on local port **2224**. Likewise, **aux2** displays anything on port **2225**. The **files** channel shows any files currently in the **~/static** directory.



A purple test page for 'UDEMA' appears. Wonderful. It's working.

30. Test if there are any running docker containers.

```
student@bchd:~$ sudo docker ps -a

CONTAINER ID    IMAGE     COMMAND         CREATED             STATUS              PORTS                    NAMES
b63ccb7ee966    webby     "/webserver"    About a minute ago  Up About a minute   0.0.0.0:2224->8888/tcp   mywebby
```

31. Test how many images exist in your current repository. If this is your first time through, you will likely only have one image, which is your newly created web server that is LIVING IN A CONTAINER! So neat.

```
student@bchd:~$ sudo docker images

REPOSITORY          TAG                 IMAGE ID            CREATED             SIZE
webby               latest              cadec56e4f74        2 minutes ago         93.2 MB
```

32. Do a docker version check to find out what version you're running.

```
student@bchd:~$ docker --version

Docker version 20.10.12, build 20.10.12-0ubuntu4
```

33. Our next task is to upload this image that we made into a trusted (SSL) Image Registry. This registry is called ***bchd.registry**. The following graphic will explain the process.

```
                          +------------>
   Upload your container to a registry that K8s trusts
                       +----------------------+
                       |                      |
+----------+   +--------+----------+   +------v-------+
|  Local   |   | Scratch Container |   | bchd.registry|
|  Webby   |   |  With Webby Files |   |   Registry   |
|  Files   |   |      Loaded       |   |              |
+----------+   +-------------------+   +--------------+
```

34. Now we can set up our own local docker registry. This is a handy little trick if you want to work on developing images locally. For now, we can just easily build our own docker registry playground.

```
student@bchd:~$ sudo docker run -d -p 2345:5000 registry:2
```

35. Next let's modify our bchd host's **/etc/hosts** file to be aware of the FQDN ***bchd.registry*** by pointing it to localhost as follows.

```
student@bchd:~$ sudo sed -i '/127.0.0.1/!b;/bchd.registry/!s/$/ bchd.registry/' /etc/hosts
```

This adds **bchd.registry** to the right of **localhost** on the first line. If already there, it will NOT add it again.

```
127.0.0.1 localhost bchd.registry

# The following lines are desirable for IPv6 capable hosts
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhost
```

36. Your bchd server is running an nginx reverse proxy to handle all of your Kubernetes commands, and directing them to the appropriate destination. Since we just added a local docker registry, we need to make our nginx server aware of our docker registry that we just brought on line. Push the following **nginx** configuration into place.

```
student@bchd:~$ sudo cp ~/mycode/config/bchd.registry /etc/nginx/sites-enabled/reg && batcat ~/mycode/config/bchd.registry && batcat /etc/nginx/sites-enabled/reg

server {
    listen bchd.registry;
    client_max_body_size 100M;

    location / {
        proxy_pass http://127.0.0.1:2345;
        }
    }
```

37. Now reload the **nginx** configuration to activate the changes.

```
student@bchd:~$ sudo nginx -s reload
```

38. Tag your new image with the FQDN of bchd.registry. This will make the push of your image go directly to your registry that is hosted on bchd.registry.

```
student@bchd:~$ sudo docker tag webby bchd.registry/webby
```

> If you come to an error with creating the docker tag, check to make sure you are not using capital letters. You will receive a parsing error.

39. Okay, now push your new image up to the **bchd.registry** registry.

```
student@bchd:~$ sudo docker push bchd.registry/webby
```

40. If your command successfully pushes to the registry, the output will look like the following:

```
The push refers to repository [bchd.registry/webby]
a58c1cd5ebb3: Pushed
feec4fa54ecc: Pushed
latest: digest: sha256:70b7f523e5e160758811c6c306ab385f604ba5119f9c99cb6429cdb786522c80 size: 740
```

41. List your current docker images.

```
student@bchd:~$ sudo docker image ls

REPOSITORY                    TAG       IMAGE ID       CREATED         SIZE
webby                         latest    9360b4b7f426   9 minutes ago   94MB
bchd.registry/webby           latest    9360b4b7f426   9 minutes ago   94MB
registry                      2         b8604a3fe854   2 days ago      26.2MB
```

42. Stop your webby container from running.

```
student@bchd:~$ sudo docker stop mywebby

mywebby
```

43. Great job! That's it for this lab.

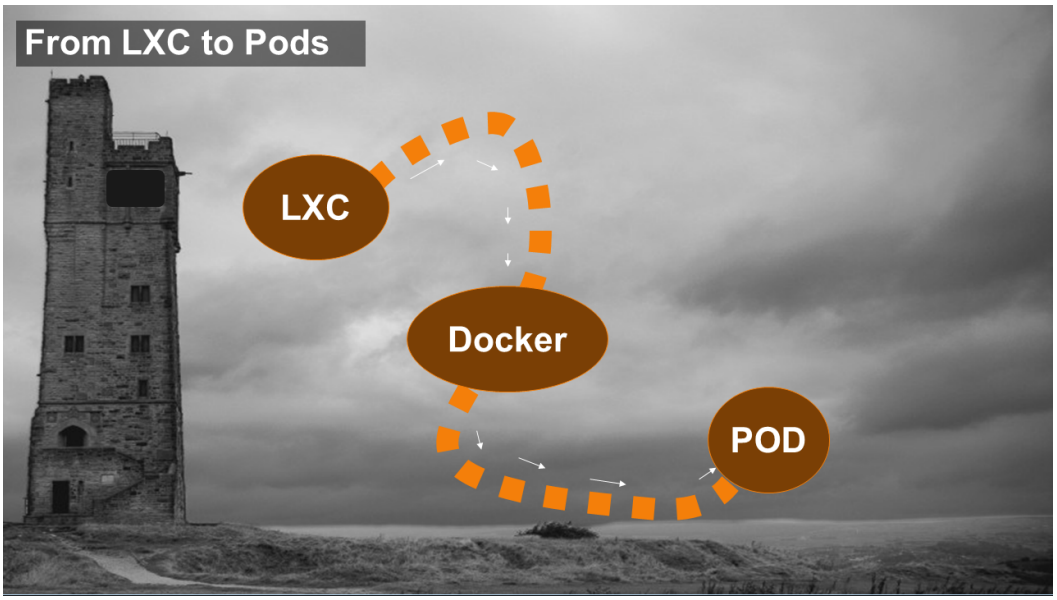
Further Reading: What are containers?

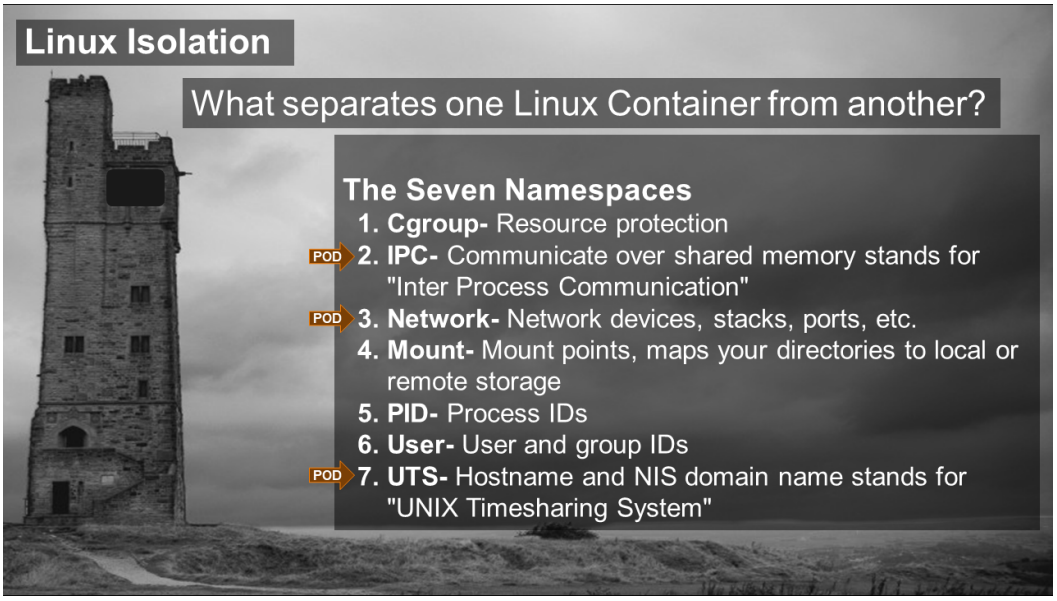
According to docker: https://www.docker.com/what-docker
Another Docker Blog: https://www.docker.com/what-container
According to Amazon: https://aws.amazon.com/what-are-containers/
Wikipedia: https://en.wikipedia.org/wiki/Docker_(software)
According to techcrunch: https://techcrunch.com/2016/10/16/wtf-is-a-container/

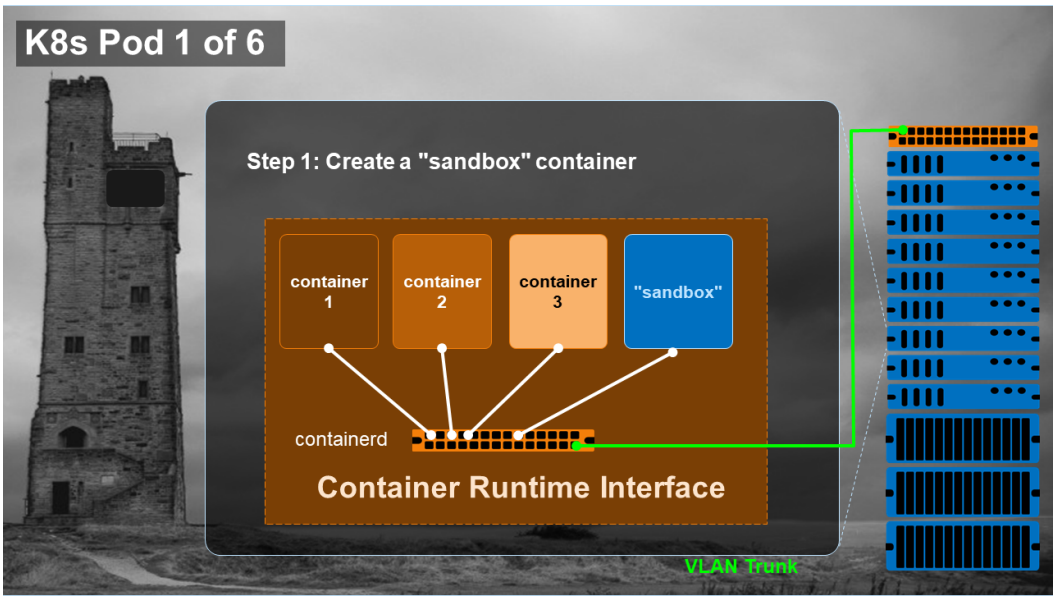
Click for Code to Catch Up!

```
# RUN THE FIRST FOUR STEPS ONE AT A TIME
cd ~
groups student
sudo usermod -aG docker $USER
groups student
sudo -i -u $USER

# Cut and PASTE the remainder and let it run
sudo apt install -y golang
go version
mkdir greeter
cd greeter
cat <<EOF > hello.go
package main
import "fmt"
func main() {
    fmt.Println("hello world")
}
EOF
go run hello.go
go mod init greeter
go build
ls
./greeter
cd ~
git clone https://github.com/alta3/webby.git
cd ~/webby
export GO111MODULE="on"
go mod init webby
go get -u github.com/stripe/stripe-go/v72
go get .
CGO_ENABLED=0 GOOS=linux go build -a -installsuffix cgo -o webserver .
wget https://static.alta3.com/courses/kubernetes/files/scratch -O Dockerfile
cat Dockerfile
sudo docker build -t webby -f Dockerfile .
sudo docker run -p 2224:8888 --name mywebby -d webby
curl localhost:2224
sudo docker ps -a
sudo docker images
docker --version
sudo docker run -d -p 2345:5000 registry:2
sudo sed -i '/127.0.0.1/!b;/bchd.registry/!s/$/ bchd.registry/' /etc/hosts
sudo wget https://labs.alta3.com/courses/kubernetes/bchd-reg -O /etc/nginx/sites-enabled/reg && cat /etc/nginx/sites-enabled/reg
sudo nginx -s reload
sudo docker tag webby bchd.registry/webby
sudo docker push bchd.registry/webby
sudo docker image ls
sudo docker stop mywebby
```

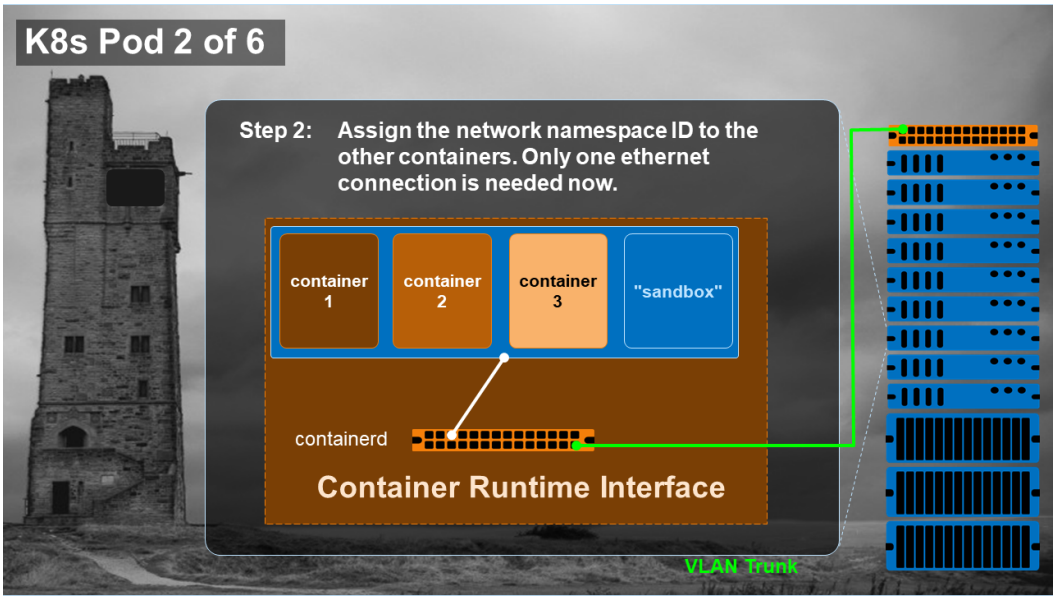
7. K8S Pods And Control Plane

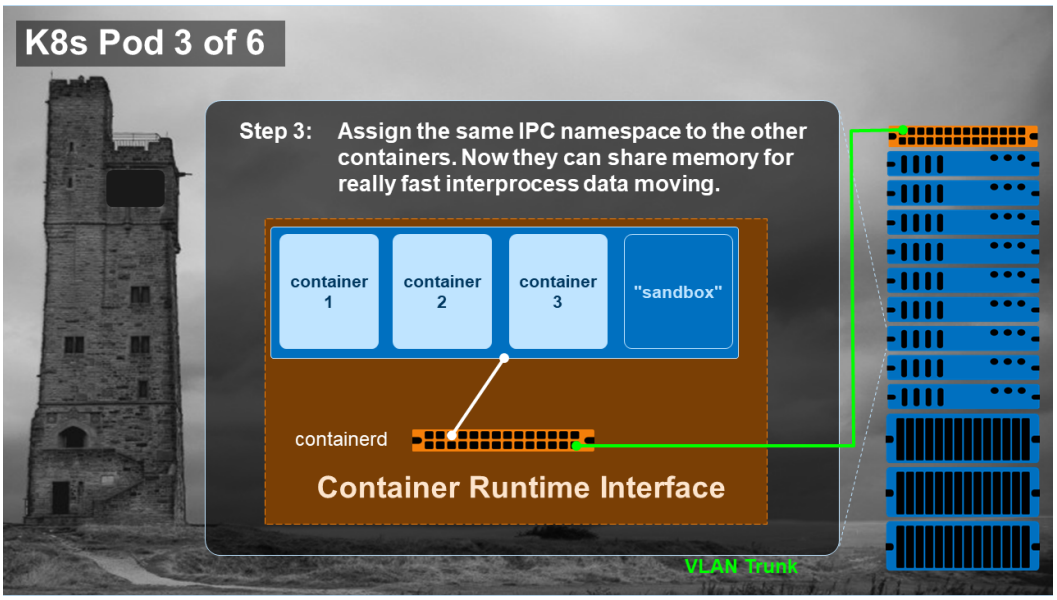


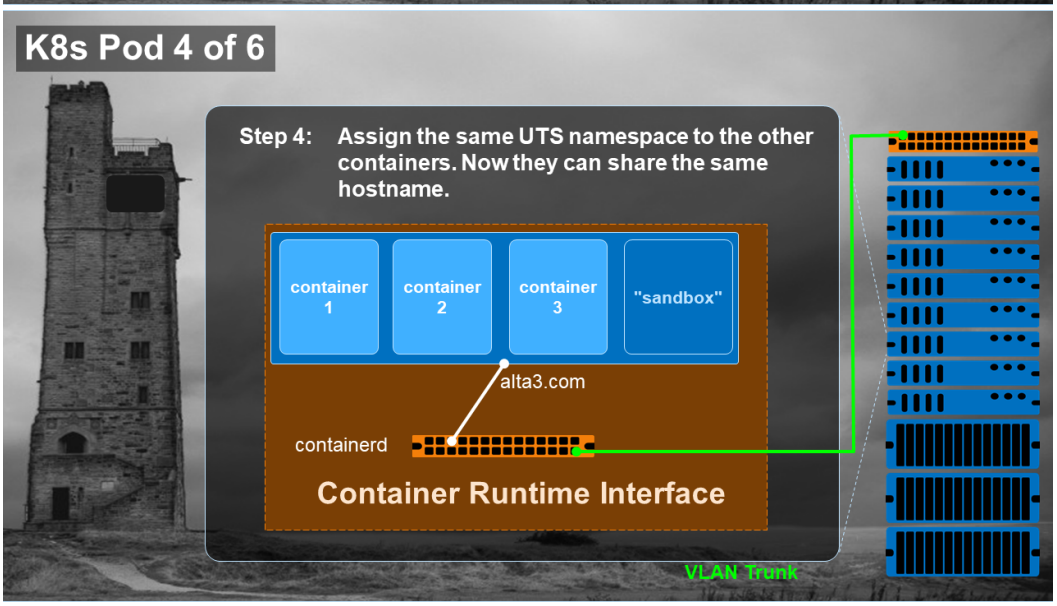




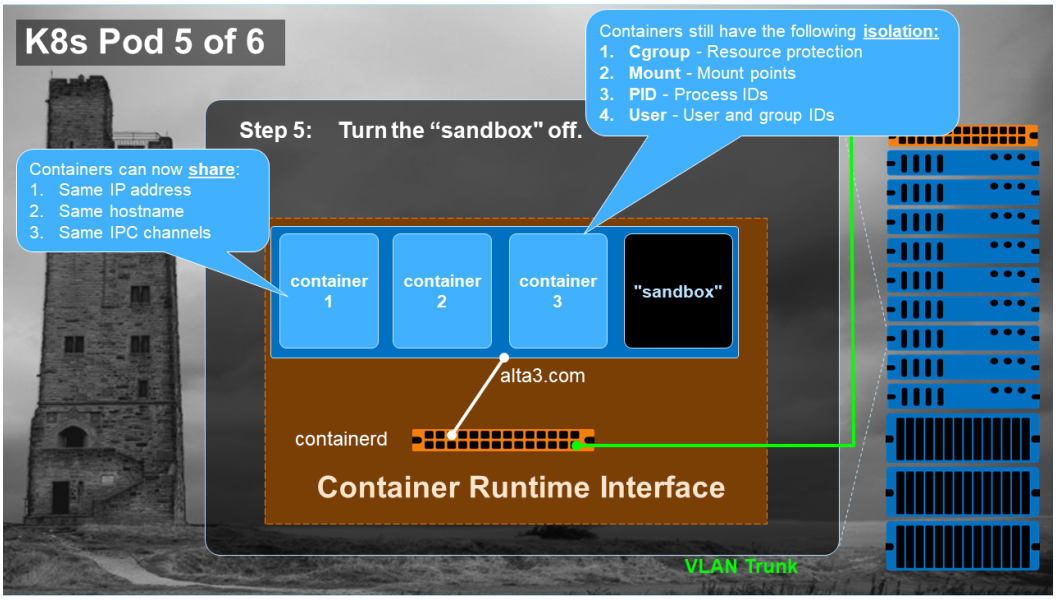
Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

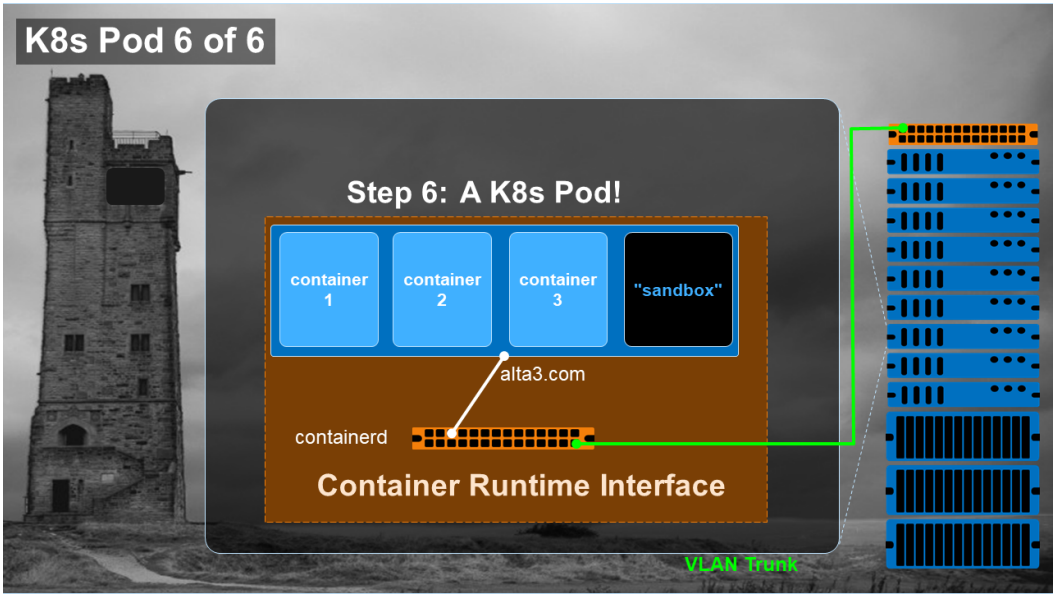


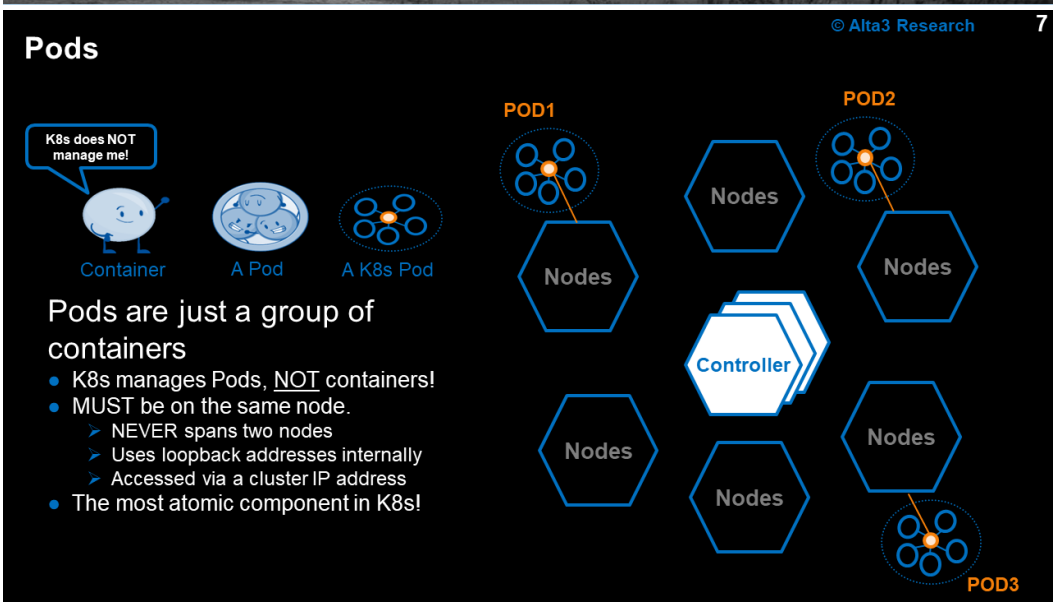




Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com



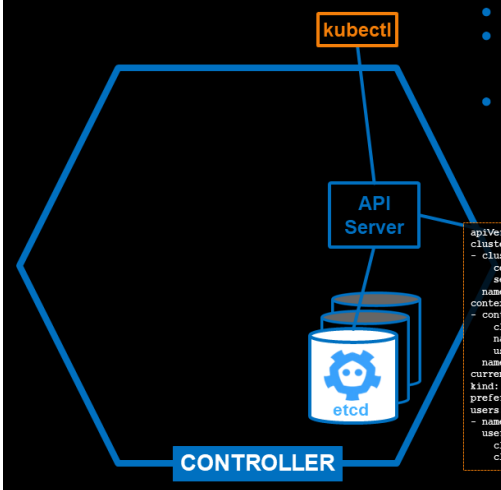




Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

kubectl

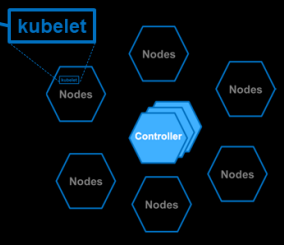
© Alta3 Research 8



kubectl

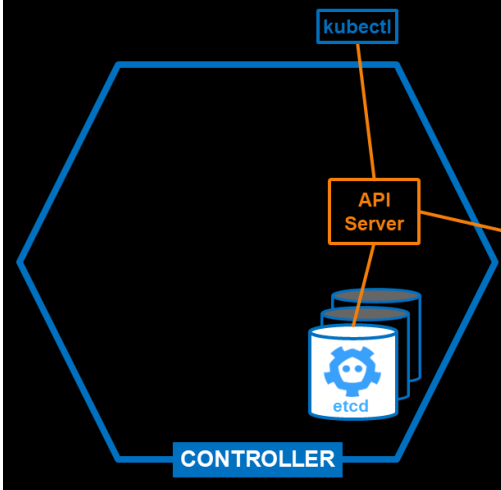
- CLI for K8s clusters
- Installs on Ubuntu, Homebrew, Macports, Powershell, Chocolatey, and Red Hat.
  - https://kubernetes.io/docs/tasks/tools/install-kubectl/
- kubeconfig file
  - Allows kubectl to find and access a K8s cluster

```
apiVersion: v1
clusters:
- cluster:
    certificate-authority: fake-ca-file
    server: https://1.2.3.4
  name: development
contexts:
- context:
    cluster: development
    namespace: frontend
    user: developer
  name: dev-frontend
current-context: dev-frontend
kind: Config
preferences: {}
users:
- name: developer
  user:
    client-certificate: fake-cert-file
    client-key: fake-key-file
```



The API Server

© Alta3 Research 9

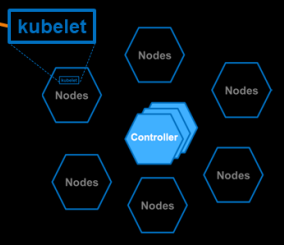


The API Server

The central management entity.
The only component that directly connects to etcd.

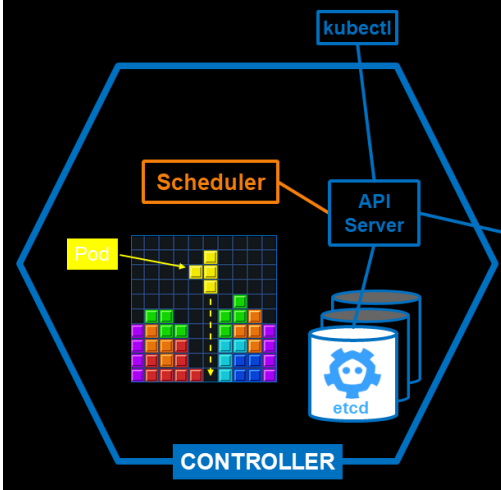
Core functionality is truly as an “API”

- **External:** via kubectl
- **Nodes:** via kubelet which runs on each node
- **Persistent state of objects via:** etcd



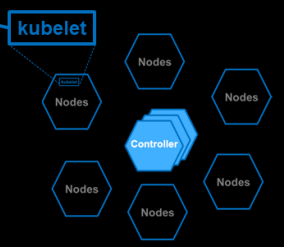
Scheduler

© Alta3 Research 10



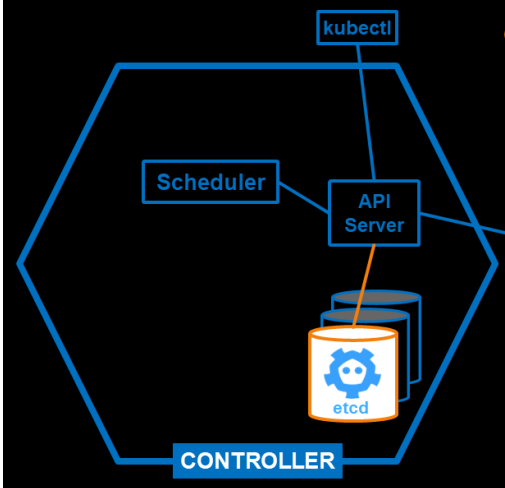
Schedules Pods on nodes

- Plays “Tetris” on all nodes
- Once the Pod is placed on a node, the scheduler is done.
- Kubelet then takes over to deploy and observe the Pod



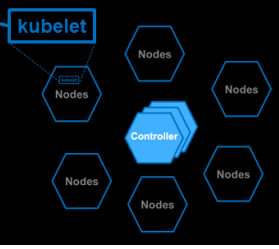
etcd

 11



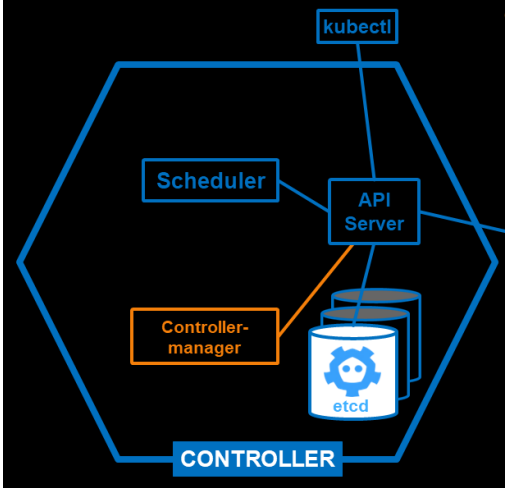
etcd

- etcd is a distributed key-value store
- The primary datastore of Kubernetes
- Stores and replicates all Kubernetes cluster states
- Runs in high availability mode (3 of them)
  - Requires a quorum for the following:
    - Elect a new ETCD Member
    - Update the datastore



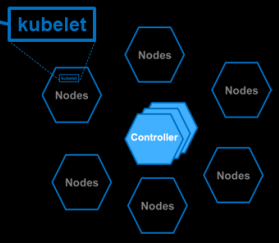
Controller-manager

 12



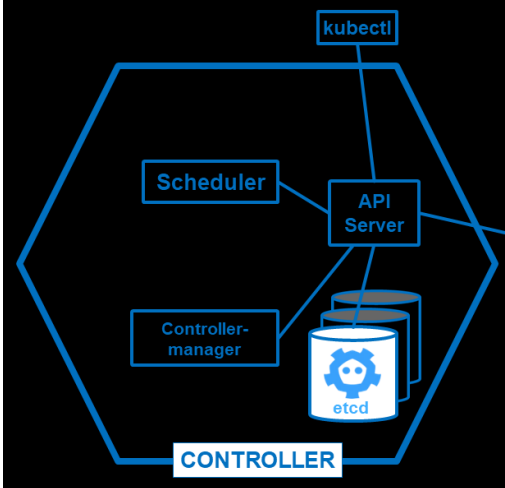
Controller-manager

- A daemon that embeds controllers inside the master node, such as:
  - replication controller
  - endpoints controller
  - namespace controller
  - serviceaccounts controller



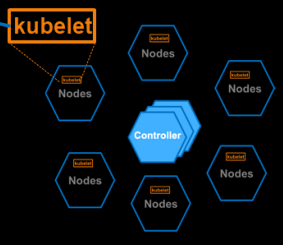
kubelet

 13



kubelet

- Focused on running containers
- Runs on all nodes
- Pods are defined by a JSON or YAML file
  - Called a Pod manifest
- Has an internal HTTP server.
  - Read-only view on port 10255
  - Kubelet URLs that you can curl
    - /health – health check
    - /pods
    - /spec



Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

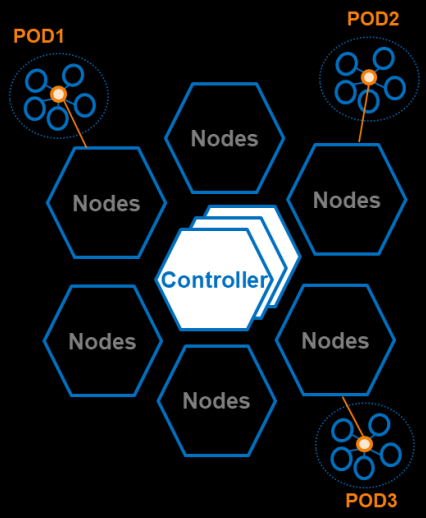
 14

Container Runtime

(Used to be Docker)

- Docker Client (kubelet handles this)
  - *"docker build"*
  - *"docker pull"*
  - *"docker run"*
- Container Runtime (runs on each node)
  - CRI (Interface)
  - Containers
  - Images
    - A read-only template with instructions for creating a container.
    - An image may be based on another image
    - Can be ready-made
- Registry (external to K8s)

A K8s Pod



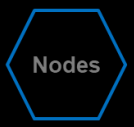
 15

What "Docker Deprecation" for Kubernetes Means

- *dockershim* is no longer being maintained by K8s. This previously provided CRI support to Docker.
- It enabled all the unnecessary features below that K8s doesn't even use or need.

dockershim

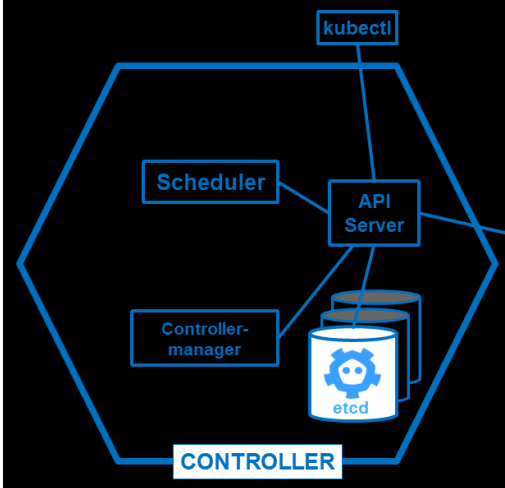
CLI | Volumes | API | Network | Server



Container Runtime

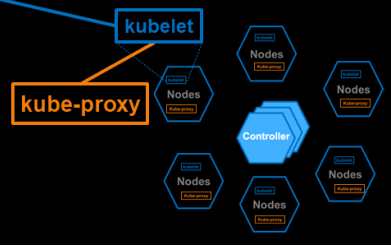
 16

kube-proxy



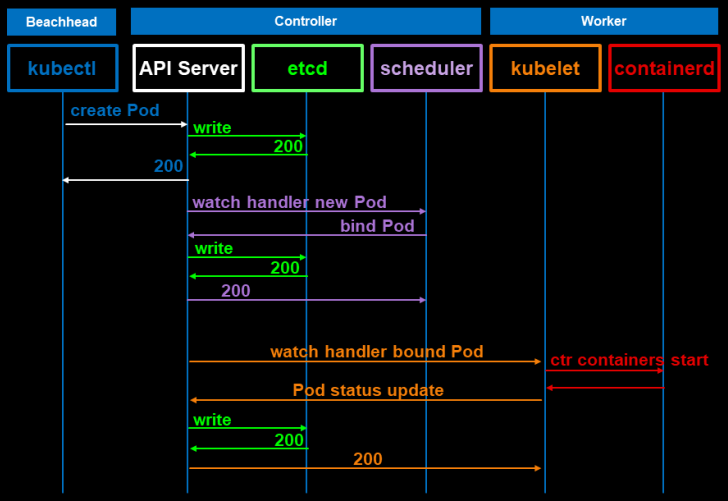
kube-proxy

- Runs on each node
- Was an actual proxy at one time, but no longer!
- In K8s 1.2, the kube-proxy gained access to iptables- therefore, the ability to delegate the actual proxy work by:
  - Inserting netfilter rules into the kernel space
  - Netfilters actually perform the redirect service
  - Packet forwarding stays in kernel space (FAST)
  - Kube-proxy's job is relegated to keeping the netfilter rules in sync.



 17

Creating a Pod



Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

8. Deploy Kubernetes using Ansible

Lab Objective

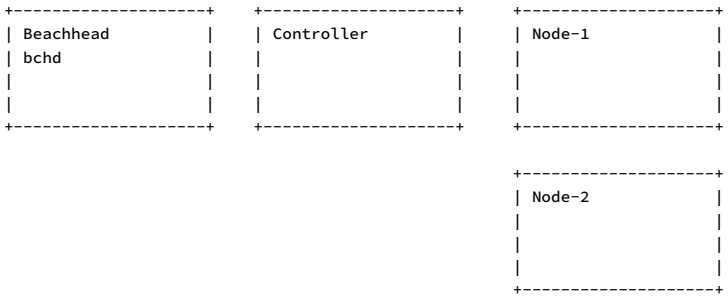
The objective of this lab is to learn the Kubernetes components you will be working with. We will make sure we have IP connectivity to all of them and familiarize ourselves with the components that constitute the underlying Kubernetes cloud, the host IP addresses, and the host names. The lab environments you will be working in leverage the Alta3 Cloud Environment Services (ACES) (https://alta3.com/) to streamline provisioning of the computer infrastructure required to bootstrap a Kubernetes cluster.

The basic working components of Kubernetes are outlined here:
https://kubernetes.io/docs/concepts/overview/components/

Now it is time to deploy Kubernetes! To do this you will use an Ansible Playbook designed by Alta3. Ansible is an open source automation tool designed to automate the deployment of configurations and software. It's actually perfect for *deploying Kubernetes*.

You can find additional information about Ansible here:
https://alta3.com/overview-ansible

Let's take a look at your current environment. After the playbook runs, the following configuration will be observed.



Questions

How many machines are running?
Four. Beachhead (your terminal), one controller node, and two worker nodes.

What are the IPs of each machine?
This will become clear once the playbook runs.

Procedure

1. We want to verify connectivity throughout our environment. Never used fping? https://fping.org/ is a program to send ICMP echo probes to network hosts. It's similar to ping, but enjoys much better performance when pinging multiple hosts.

   ```
   student@bchd:~$ fping -A controller  node-1 node-2
   ```

   The output will be similar to the below.

   ```
   10.5.203.188 is alive
   10.13.108.236 is alive
   10.8.217.4 is alive
   ```

2. Let's initialize your current Kubernetes environment (this may take a while, somewhere around eight minutes). If you are interested in what changes occurred, look at the playbook readout. In short, running the playbook sets up your networking, puts certs in the right location, and deploys etcd, nodes, controllers, services, and load balancer.

3. Move into the **~/git/kubernetes-the-alta3-way** directory.

   ```
   student@bchd:~$ cd ~/git/kubernetes-the-alta3-way
   ```

4. Run the Ansible playbook. Ansible is a tool that allows users to describe the state they want to achieve in code using a YAML document called a 'playbook.' Ansible then works to ensure that the state described in the playbook is achieved on the target machines. In short, Ansible makes it easy for anyone to write complex installation routines. The playbook you are about to run, **main.yml**, will install Kubernetes. The inventory file, **hosts.yml**, instructs Ansible to install Kubernetes on specific targets.

```
student@bchd:~/git/kubernetes-the-alta3-way$ ansible-playbook main.yml
```

> The output will look similar to the example below...

```
PLAY [bootstrap] ***************************************************************************************

TASK [Gathering Facts] *********************************************************************************
ok: [controller]
ok: [node-1]
ok: [node-2]

TASK [bootstrap : raw apt update 1] ********************************************************************
changed: [controller]
```

5. If the playbook ends with some **red fails**, let the instructor know.
6. While you are waiting for that to finish up, if you have a GitHub account, log in and visit https://github.com/alta3/kubernetes-the-alta3-way and click on the star at the top right of the page.
7. To learn more about tmux and your environment, please visit our quick demonstration on tmux here: https://youtu.be/xKJfbs9eSug
8. Return to the home directory.

```
student@bchd:~/git/kubernetes-the-alta3-way$ cd ~
```

9. Kubectl is a command line tool for controlling Kubernetes clusters. Issue **kubectl get nodes** list the nodes.

```
student@bchd:~$ kubectl get nodes

NAME     STATUS   ROLES    AGE    VERSION
node-1   Ready    <none>   133m   v1.23
node-2   Ready    <none>   133m   v1.23
```

10. Use the **kubectl describe nodes** command to view detailed information about the nodes. Your instructor can help you parse back through this information, as it may scroll off your screen. You will learn more about the **get** and **describe** commands in subsequent labs.

```
student@bchd:~$ kubectl describe nodes
```

> To scroll using tmux, first press CTRL + B, then [and if you've done it correctly, a yellow screen buffer will appear in the upper right corner. You can now use the up and down arrow to scroll, and q to quit.

11. This will come in handy for you. Adding to your .bashrc file will allow for autocompletion of the kubectl command.

```
student@bchd:~$ echo 'source <(kubectl completion bash)' >> ~/.bashrc
```

12. Now re-source your **.bashrc** file.

```
student@bchd:~$ source ~/.bashrc
```

13. Next lets verify that this works as expected by creating our own pod, using our own image. We will discuss this in more depth later on.

```
student@bchd:~$ batcat ~/mycode/yaml/my_first_pod.yaml
```

Click here to view the contents of **my_first_pod.yaml**

14. Run the new manifest.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/my_first_pod.yaml
```

15. Next we will make sure that the Pod exists and has launched successfully.

```
student@bchd:~$ kubectl wait --for condition=Ready --timeout 30s pod/myfirstpod

student@bchd:~$ kubectl get pods

NAME         READY   STATUS    RESTARTS   AGE
myfirstpod   1/1     Running   0          19s
```

16. Success! You won't need that pod anymore, so go ahead and delete it.

```
student@bchd:~$ kubectl delete -f ~/mycode/yaml/my_first_pod.yaml
```

Click for Code to Catch Up!

```
{ # Deploy Kubernetes using Ansible
pushd $HOME/git/kubernetes-the-alta3-way
ansible-playbook main.yml
popd
kubectl get nodes
kubectl apply -f "${HOME}/mycode/yaml/my_first_pod.yaml"
kubectl wait --for condition=Ready --timeout 30s pod/myfirstpod
kubectl get pods
kubectl delete -f "${HOME}/mycode/yaml/my_first_pod.yaml"
} || echo "Execution stopped, see above error"
```

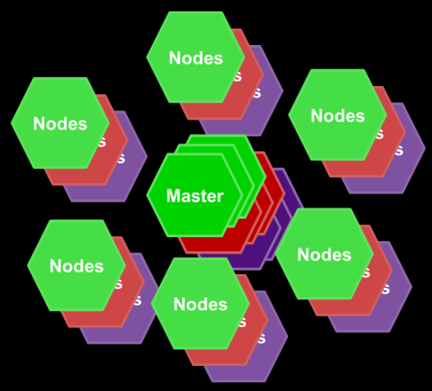
9. Namespaces

k8s namespace defined

 2

Namespaces provide features such as:

- A scope for names so that names of resources must only be unique within a namespace, not across all namespaces.
- This allows names to be reused from one namespace to the next.
- Namespaces are a way to enforce cluster resources using a resource quota.



kubectl get command

 3

```
kubectl get namespaces
```

Kubernetes starts with four initial namespaces:

default - The default namespace for objects with no other namespace

kube-system - The namespace for objects created by the Kubernetes system

kube-public - This namespace is created automatically
- Readable by all users
- Mostly reserved for cluster usage
- Used when resources should be visible and readable publicly throughout the whole cluster.

kube-node-lease - The namespace where Nodes' Lease objects are stored (a Lease controls "node heartbeats," which are used for node monitoring).

Creating, Updating and Destroying K8s Objects

 1

This will create an "object"
```
kubectl create -f object.yaml
```

This can create or change existing "object"
```
kubectl apply -f object.yaml
```

Interactive edits are possible rather than editing the local file
```
kubectl edit <resource> <obj>
```

You may also use the same YAML to delete an object
```
kubectl delete -f object.yaml
```

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

10. Yaml

YAML

© Alta3 Research 1

Scalar Types

```
key: value
second key: a different value
a_number: 156
boolean: true
null_value: null
```

YAML

© Alta3 Research 2

Multiple-line strings

```
A_literal_block: |
  This is a really long value! And every
  line break will be preserved.

  This value will last until the text is
  "dedented".
Folded_text: >
  All of the text will
  have all newlines replaced
  with a single space.
```

YAML

 3

Collections

```
my_collection:
  key: value
  second key: a different value
  other values:
    number: 145
    boolean: true
    null_value: null
```

YAML

 4

```
my_sequence:
- ItemA
- ItemB
- aNestedCollectOfGroceries:
    eggs: fried
    bacon: crispy
- key: value
```

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

11. Create Config Pods

Defining a Pod

 1

- A Pod is a collection of one or more containers
- A Pod cannot span nodes
  - All containers within a Pod land on the same node
- All containers within a Pod will share:
  - Linux Network namespace (share IP addresses)
  - UTS namespace (fancy term for sharing the same hostname)
  - IPC namespace (shares the same interprocess communication channels)

Creating Pod Manifest

 2

Using a Pod manifest nginx.1.18.yaml file

```
apiVersion: v1
kind: Pod
metadata:
  name: examplepod
  namespace: test
spec:
  containers:
  - name: nginx
    image: nginx:1.18.9
    ports:
    - containerPort: 8080
      name: http
      protocol: TCP
  - name: webby
    image: registry.bchd/alta3-webby
    ports:
    - containerPort: 8888
      name: webby-port
      protocol: TCP
```

Delete Pods

 3

Delete the Pod by name

```
kubectl delete pods nginx
```

Delete using the YAML file that created the Pod in the first place

```
kubectl delete -f nginx-pod.yaml
```

12. Create and Configure Basic Pods

CKAD Objective

- *Create and configure basic Pods*

Lab Objective

The purpose of this lab is to create a Pod manifest which will then be used to create a Pod.

You have already launched Pods, however, you have yet to actually write any **Pod Manifests**. These **Pod manifests** are what declaratively define all of the information necessary to launch a Pod. The kubectl command is used to create many different Kubernetes resource objects, not just Pods.

To see the list of Kubernetes Resource Types, click this link:
https://kubernetes.io/docs/reference/kubectl/overview/#resource-types

Pod manifests can be written using YAML or JSON. YAML is usually preferred because it is more human-readable and has the ability to add comments. Manifests should be written in the same sense as source code. Manifests include some key fields and attributes, such as: a metadata section for the pod and label description, a spec section for describing volumes, and a list of containers that will all run in the Pod.

If the concept of Pods is confusing, check out the following documentation:
https://kubernetes.io/docs/concepts/workloads/pods/pod/

For assistance with writing manifests, here are some more resources for you to peruse through manifest examples:
https://static.alta3.com/files/Manifests.pdf

An extended pod manifest example:
https://static.alta3.com/files/ExtendedManifest.pdf

In an attempt to make this lab more difficult, a few steps have been omitted. However, all answers are documented at the bottom of the lab.

Questions

What is the difference between a Pod and a Pod manifest?
A Pod is what is created by Kubernetes. To create a Pod, a user can use the command line or create a manifest, which is typically written in YAML

Can you name the steps (without looking) to deploy your Pod, from start to finish?
Create a manifest, then issue "kubectl create -f "

Recall what's included in a Pod manifest. What can the value of "kind" be?
The value of "kind" is the object you want Kubernetes to create. Some of the ones we will study in this course include: "Pod", "Deployment", "ReplicationController" (Manages Pods), "DeploymentController" (Manages Pods), "StatefulSets", "DaemonSets", "Services", "ConfigMaps", and "Volumes".

Procedure

1. Run the setup.

```
student@bchd:~$ setup create-and-configure-basic-pods
```

2. Let's begin by taking a quick look at what contexts are available to our kubectl client. Remember, a context is a reference to a cluster, namespace, and user.

```
student@bchd:~$ kubectl config get-contexts
```

3. Set kubectl so it controls the namespace `default`, within our cluster, as the user `admin`. We gave this configuration the name, `kubernetes-the-alta3-way` context.

```
student@bchd:~$ kubectl config use-context kubernetes-the-alta3-way

Switched to context "kubernetes-the-alta3-way".
```

4. A "manifest" is written in YAML and documents, "what you want set up". Within a manifest, Kubernetes "keys" and "values" document what you want created, whereas YAML is the "structure" that organizes it all. Now is a good time to review the rules of YAML: http://yaml.org/spec/1.2/spec.html

5. A pod manifest *must* contain certain keys. Since this is YAML, these keys can be in any order, but the classic approach is alphabetical order- this approach is convenient because alphabetical order makes sense for reading the values.

```
apiVersion:
kind:
metadata:
spec:
```

> **apiVersion:** is a required `key` field. It defines the Kubernetes API you are using.
> **kind:** For this lab, the value field will be either `Pod` or `Deployment`.
> **metadata:** A required `key` that helps uniquely identify the object. Subkeys are `name`, `uid`, `labels`, and `namespace`.
> **spec:** Specifies the desired state of an object. If a spec is deleted, the object will be purged from the system.
> **status:** PURELY FYI - A system generated value that shows the actual state of the object at the current time.

6. Focus on that key `kind:`. The `kind:` key describes the object you want Kubernetes to create. Curious what the **entire** list of objects is? Issue the following command. The far right column reflects the 'value' you could place to the right of the key `kind:` within your manifest.

```
student@bchd:~$ kubectl api-resources
```

7. Time to experiment with writing your own pod manifest. **Every line in this file that has a `#` needs to have the value from the next step, step 6 added into this file.** For example: The second line here will end up being "apiVersion: v1". The *v1* comes from the first line's value in the next step.

```
student@bchd:~$ vim ~/mycode/yaml/simpleservice-empty.yaml
```

Click here to view the contents of **simpleservice-empty.yaml**

> **BEFORE** you exit the simpleservice-empty.yaml file, edit the above empty manifest using vim according to the values shown below. REMOVE the hashtag and comments and replace with the data below. *The completed YAML document can be found at the bottom of this lab, but try to create it yourself. It will force you to study YAML notation.*

- apiVersion = v1
- kind = Pod
- metadata.name = simpleservice
- metadata.labels.name = simpleservice-web
- spec.containers[0].name = simpleservice-web
- spec.containers[0].image = mhausenblas/simpleservice:0.5.0
- spec.containers[0].ports[0].name = web
- spec.containers[0].ports[0].containerPort = 9876
- spec.containers[0].ports[0].protocol = TCP

> One you've finished updating `simpleservice.yaml`, save with `:wq` and exit.

8. The image referenced above, "simpleservice" is a container image. You can read about the image and the "simple API service" it provides: https://hub.docker.com/r/mhausenblas/simpleservice/

9. Answer the following question:

   - **Why is access on port 9876?**
     - *Per the documentation on the "simpleservice" image, this is the port the service is listening on.*

10. Boot a pod using the pod manifest you created.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/simpleservice-empty.yaml

pod/simpleservice created
```

11. At the CLI, use `describe` to display pod details (more details than the `get` command).

```
student@bchd:~$ kubectl describe pod simpleservice
```

12. Delete the `simpleservice` pod by referencing the same manifest.

```
student@bchd:~$ kubectl delete -f ~/mycode/yaml/simpleservice-empty.yaml

pod "simpleservice" deleted
```

13. Think back to the first lab. You took approximately 30 minutes of your life to build and run webby. It would be better if we described running our webby image in a Pod within a YAML document. This is easily version controlled and commented. We learned this technique at the top of the lab. Open the file and fill out the template!

```
student@bchd:~$ vim ~/mycode/yaml/webby-pod01-empty.yaml
```

Click here to view the contents of **webby-pod01-empty.yaml**

When you're finished filling out the template, Save and exit with `:wq`.

14. Run the pod using the YAML file you just created.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/webby-pod01-empty.yaml

pod/webservice01 created
```

15. At the CLI, use `describe` to display webby details.

```
student@bchd:~$ kubectl describe pod webservice01
```

16. Delete the webby pod.

```
student@bchd:~$ kubectl delete -f ~/mycode/yaml/webby-pod01-empty.yaml
```

17. **That's it for this lab!** For answer keys to our pod templates, see below.

18. ANSWER - simpleservice-empty.yaml

    Click

```
 batcat mycode/yaml/simpleservice.yaml
──────┬──────────────────────────────────────────────
      │ File: mycode/yaml/simpleservice.yaml
──────┼──────────────────────────────────────────────
   1  │ apiVersion: v1
   2  │ kind: Pod
   3  │ metadata:
   4  │   name: simpleservice
   5  │   labels:
   6  │     name: simpleservice-web
   7  │ spec:
   8  │   containers:
   9  │   -
  10  │     name: simpleservice-web
  11  │     image: mhausenblas/simpleservice:0.5.0
  12  │     ports:
  13  │     -
  14  │       name: web
  15  │       containerPort: 9876
  16  │       protocol: TCP
──────┴──────────────────────────────────────────────
```

19. ANSWER - webby-pod01-empty.yaml

    Click

```
batcat mycode/yaml/webby-pod01.yaml
──────┬──────────────────────────────────────────────
      │ File: mycode/yaml/webby-pod01.yaml
──────┼──────────────────────────────────────────────
   1  │ apiVersion: v1
   2  │ kind: Pod
   3  │ metadata:
   4  │   name: webservice01
   5  │ spec:
   6  │   containers:
   7  │   - name: webby
   8  │     image: registry.gitlab.com/alta3/webby
   9  │     ports:
  10  │     - name: web
  11  │       containerPort: 8888
  12  │       protocol: TCP
  13  │
──────┴──────────────────────────────────────────────
```

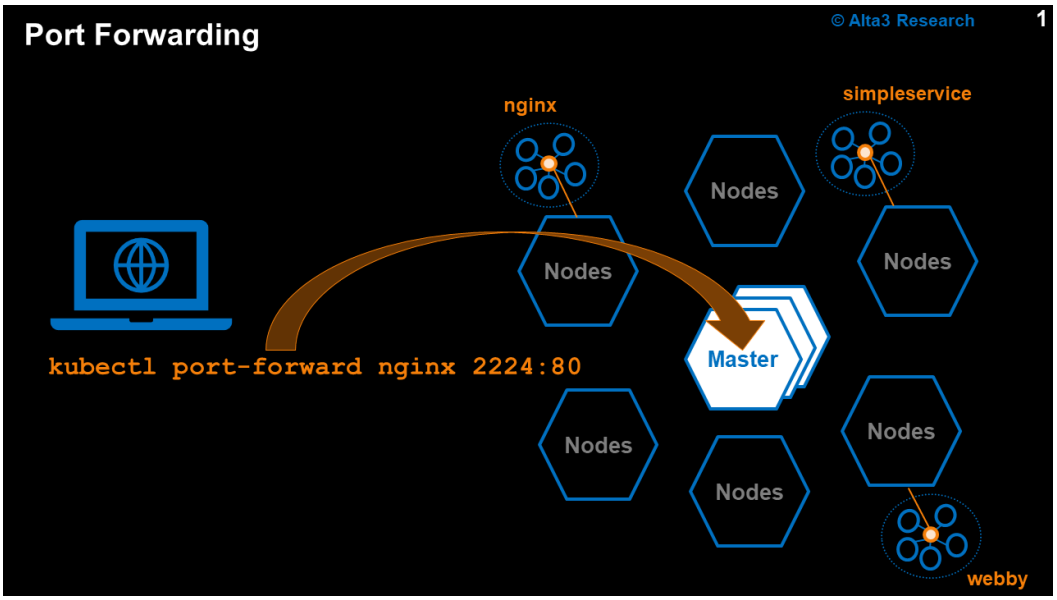
API Versions and Deprecations

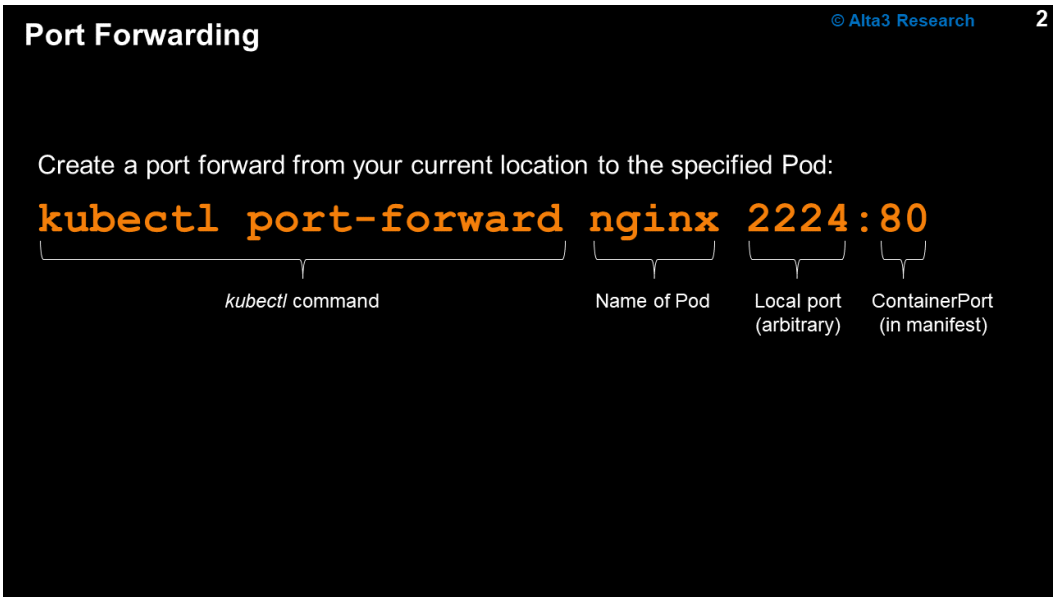
API Version	Track
v1alpha1	Alpha (experimental)
v1beta1	Beta (pre-release)
v1	GA (generally available, stable)

API Versions and Deprecations

Release	API Versions	Preferred/ Storage Version	Notes
X	v1alpha1	v1alpha1	
X+1	v1alpha2	v1alpha2	•v1alpha1 is removed, "action required" relnote
X+2	v1beta1	v1beta1	•v1alpha2 is removed, "action required" relnote
X+3	v1beta2, v1beta1 (deprecated)	v1beta1	•v1beta1 is deprecated, "action required" relnote
X+4	v1beta2, v1beta1 (deprecated)	v1beta2	
X+5	v1, v1beta1 (deprecated), v1beta2 (deprecated)	v1beta2	•v1beta2 is deprecated, "action required" relnote
X+6	v1, v1beta2 (deprecated)	v1	•v1beta1 is removed, "action required" relnote
X+7	v1, v1beta2 (deprecated)	v1	
X+8	v2alpha1, v1	v1	•v1beta2 is removed, "action required" relnote
X+9	v2alpha2, v1	v1	•v2alpha1 is removed, "action required" relnote
X+10	v2beta1, v1	v1	•v2alpha2 is removed, "action required" relnote
X+11	v2beta2, v2beta1 (deprecated), v1	v1	•v2beta1 is deprecated, "action required" relnote
X+12	v2, v2beta2 (deprecated), v2beta1 (deprecated), v1 (deprecated)	v1	•v2beta2 is deprecated, "action required" relnote •v1 is deprecated, "action required" relnote
X+13	v2, v2beta1 (deprecated), v2beta2 (deprecated), v1 (deprecated)	v2	
X+14	v2, v2beta2 (deprecated), v1 (deprecated)	v2	•v2beta1 is removed, "action required" relnote
X+15	v2, v1 (deprecated)	v2	•v2beta2 is removed, "action required" relnote
X+16	v2, v1 (deprecated)	v2	
X+17	v2	v2	•v1 is removed, "action required" relnote

13. Portforwarding



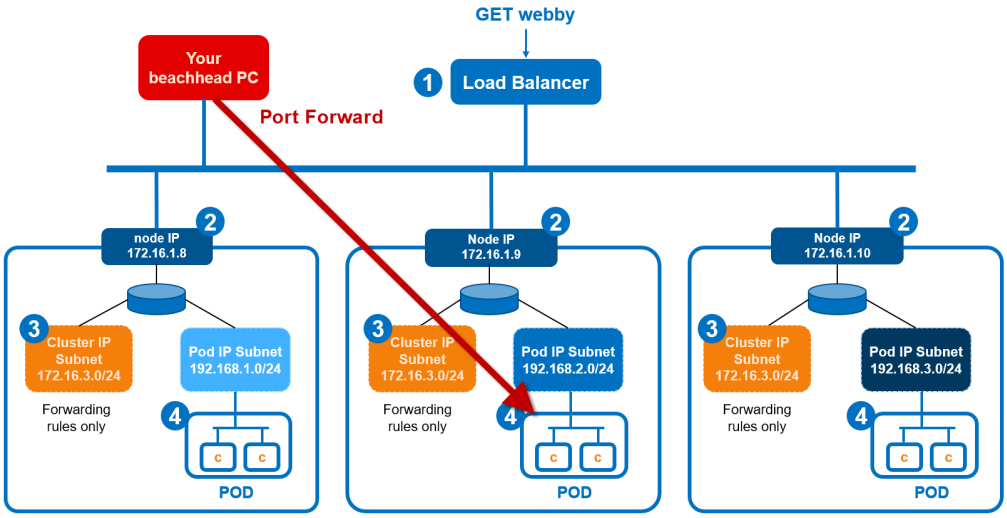


Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

14. Debugging via kubectl port-forward

Lab Objective

- Use port-forwarding support built into Kubernetes API and CLI tools.
- Create a secure tunnel from your local machine through the controller to the instance of the Pod running on one of the worker nodes.



When you need to debug a service which resides inside a pod, then port-forwarding comes to the rescue. For instance, you've got a web server and a database that are connected, but now you want to run additional commands on the db to prototype new SQL queries. The Postgres port is not exposed except within the pod namespace. This is fine for our web server and being able to access it. But now you want to get your hands on the database that's running in order to run new SQL queries. You can port forward that container's Postgres port back to your machine (local host) so we can run psql cli or pgcli in order to poke and prod at the database directly.

Let's try another example. Rather than use kubectl, you use the Kubernetes package deployment tool, Helm, to deploy an interesting app available from https://hub.helm.sh, maybe https://hub.helm.sh/charts/stable/bitcoind. With a single command line instruction to Helm, your Bitcoin wallet could be deployed into a cluster. With a port-forward, you can easily forward local traffic directly to a Kubernetes pod.

Procedure

1. Let's begin by taking a quick look at what contexts are available to our kubectl client. Remember, a context is a reference to a cluster, namespace, and user.

   ```
   student@bchd:~$ kubectl config get-contexts
   ```

2. Set kubectl so it controls the namespace `default`, within our cluster, as the user `admin`. We gave this configuration the name, `kubernetes-the-alta3-way` context.

   ```
   student@bchd:~$ kubectl config use-context kubernetes-the-alta3-way

   Switched to context "kubernetes-the-alta3-way".
   ```

3. If you haven't read about the `simpleservice` image yet, it is **strongly** recommended that you check out the GitHub repo found at https://github.com/mhausenblas/simpleservice as you will find, this image serve up several endpoints. These include `/info`, `/health`, and `/env`, to name a few. Again, it is best to check out the GitHub page and scroll through the README.md.

4. Check out the manifest `simpleservice.yaml`.

   ```
   student@bchd:~$ cat  ~/mycode/yaml/simpleservice.yaml
   ```

```
---
apiVersion: v1 # the apiVersion to use
kind: Pod
metadata:
  name: simpleservice   # name of our pod
  labels:
    name: simpleservice-web  # label on our pod (id)
spec:
  containers:
  - name: simpleservice-web  # name of the container
    image: mhausenblas/simpleservice:0.5.0  # image and version to pull from hub.docker.io
    ports:
    - name: web # port name
      containerPort: 9876 # port to direct container traffic to
      protocol: TCP # TCP or UDP
```

5. Run the `apply` command against the manifest to create the pod.

```
student@bchd:~$ kubectl apply -f  ~/mycode/yaml/simpleservice.yaml
```

6. According to the README on the simpleservice GitHub repo, a number of API endpoints should be available. The issue is that this image and its APIs will be running in a container. That container will be in a Pod with a random IP address, on some Node that also has it's own IP address. However, if we want to cut straight to that Pod, we can.

7. Open a new tmux window to use exclusively for activating port forwarding. Do this by pressing (Ctrl + b) then (Shift + ") to split the screen.

8. In the new terminal, create a tunnel from your local machine to your pod.

```
student@bchd:~$ kubectl port-forward simpleservice 2224:9876 --address=0.0.0.0

Forwarding from 0.0.0.0:2224 -> 9876
```

9. With the port forward still running, press (Ctrl + b) then the direction arrow to move your cursor between tmux session windows.

10. In the window that is **not** running your port forward, try to curl one of the simpleservice APIs from your terminal session.

```
student@bchd:~$ curl http://127.0.0.1:2224/env
```

> If you didn't know, `curl` is a command line tool that creates an HTTP GET (which is the same thing a browser sends when you browse to HTTP resources.)

11. Try another endpoint:

```
student@bchd:~$ curl http://127.0.0.1:2224/health
```

12. Type `exit` to close your current tmux window. *Note: You can always close a tmux screen by typing "exit". If you close ALL of your tmux screens, just press the "refresh" button to bring your session back!*

13. In the terminal running your port-forwarding, stop the port forwarding with (Ctrl + c)

14. Run the `delete` command against the manifest to remove the pod.

```
student@bchd:~$ kubectl delete -f ~/mycode/yaml/simpleservice.yaml
```

15. If your port-forward is stopped and your pod removed, then great job! That's it for this lab.

CHALLENGE:

Create a quick and dirty pod with the `kubectl run` command:

```
kubectl run portforwardpod --image=nginx:1.19.6
```

Run a port-forward to this Pod! `curl` the Pod to confirm it is working. You can use whatever localhost port you like, but you MUST use the correct container port. *Hint: Google for nginx port numbers!*

Click for Code to Catch Up!

```
kubectl config get-contexts
kubectl config use-context kubernetes-the-alta3-way
cat simpleservice.yaml
kubectl apply -f simpleservice.yaml
sleep 60
kubectl port-forward simpleservice 2224:9876 --address=0.0.0.0 > /dev/null 2>&1 &
curl http://127.0.0.1:2224/env
curl http://127.0.0.1:2224/health
kubectl delete -f simpleservice.yaml
```

15. Kubectl Exec

Exec

 1

Execute a command on nginx

```
kubectl exec nginx -- date
```

Get an interactive session

```
kubectl exec -it nginx -- bash
```

Copy Files

 2

Local -> Pod (only one container) INTO a container

```
kubectl cp /path/to/local/file <pod-name>:/path/to/remote/file
```

Pod (only one Container) → Local files FROM a container

```
kubectl cp <pod-name>:/path/to/remote/file /path/to/local/file
```

Local → Pod (with *more than one container*… specifically, webby)

```
kubectl cp /path/to/local/file <pod-name>:/path/to/remote/file -c webby
```

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

16. Performing Commands inside a Pod

Lab Objective

- In this lab we will go back to the kubectl `get` command, but this time utilizing it against pods.

Sorting can be a very useful thing to an administrator or manager. It's a way to troubleshoot and a way to organize information. There are also other command options such as *--sort-by*, which will make the Kubernetes journey far easier!

Organization is going to be key with any implementation.

Questions

What is the benefit of organizing your pods in a list by restart count?
Pods shouldn't have an excessive number of restarts. Those that do warrant investigation as to why they are continually being restarted.

Procedure

1. Let's begin by taking a quick look at what contexts are available to our kubectl client. Remember, a context is a reference to a cluster, namespace, and user.

```
student@bchd:~$ kubectl config get-contexts
```

2. Set kubectl so it controls the namespace `default`, within our cluster, as the user `admin`. We gave this configuration the name, `kubernetes-the-alta3-way` context.

```
student@bchd:~$ kubectl config use-context kubernetes-the-alta3-way
```

3. Get into a container to cause a purposeful restart of the pod. To start, we'll use the `apply` command. The `apply` command allows a user to create or update resources. More on this later- for now, just know we're creating a pod resource. The **-f** flag is 'file', and it is fine that the file is accessed over https. there's no need to keep files local if there is a secure place to store them remotely.

```
student@bchd:~$ kubectl create -f https://raw.githubusercontent.com/alta3/kubernetes-the-alta3-way/main/labs/yaml/nginx-pod.yaml
```

4. If you're curious what `nginx-pod.yaml` looks like, it is shown below.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.18.0
    ports:
    - containerPort: 80
```

5. Great. Now we can connect to a shell within the running container by using the `exec` command. The `sh` at the end of the command specifies that we wish to use the Bourne shell. If you get "error: unable to upgrade connection: container not found ('nginx')", wait a few seconds, as the pod is most likely still getting the container image launched.

```
student@bchd:~$ kubectl exec -it nginx -- sh
```

6. First identify the user as whom you are currently issuing commands (it should be root).

```
# whoami
```

7. List the processes running inside of the container. The */proc/* folder will contain an entry for each PID. The 1st PID is the container process itself (init).

```
# ls /proc/ | head
```

8. Imagine our container is running some really poorly written code, but it provides a critical service. The patch is in the works but for now we'll pretend that some spectacular failure occasionally happens. The take away is that if the container stops, Kubernetes should turn it back on. This is different than, say, KVM (or other hypervisors)- if we're 'inside' a Virtual Machine and shut it down, it will stay down. However, because Kubernetes focuses on service management, and shares the same kernel with the container, it's quite easy for Kubernetes to recognize the service has failed and turn it back on.

9. You issue `kill -15 1`, which should end all processes you're 'allowed' to shut down, including PID 1 (your container initialization).

```
# kill -15 1
```

10. You should be back at your host machine.

```
student@bchd:~$
```

11. Kubernetes focuses on service management so let's check on the NGINX pod. It seems that as fast as you killed the pod, Kubernetes ***restarted*** it. Even the old pod is cleaned up. You issue a command to sort the pods by restart count. You notice that the NGINX container is still running, although the RESTARTS column has been incremented by 1.

```
student@bchd:~$ kubectl get pods --all-namespaces --sort-by='.status.containerStatuses[0].restartCount'

NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE
kube-system   coredns-5d65dd49c8-jmgcj                   1/1     Running   0          36h
kube-system   calico-kube-controllers-7798c85854-dbfvh   1/1     Running   0          36h
kube-system   calico-node-8thjn                          1/1     Running   0          36h
kube-system   calico-node-hfcj8                          1/1     Running   0          36h
kube-system   calico-node-jrqrg                          1/1     Running   0          36h
kube-system   coredns-5d65dd49c8-67rgd                   1/1     Running   0          36h
default       nginx                                      1/1     Running   1          8m1s
```

12. Delete the pod to clear our cluster.

```
student@bchd:~$ kubectl delete -f https://raw.githubusercontent.com/alta3/kubernetes-the-alta3-way/main/labs/yaml/nginx-pod.yaml

pod "nginx" deleted
```

13. Confirm that it is indeed gone.

```
student@bchd:~$ kubectl get pods --all-namespaces --sort-by='.status.containerStatuses[0].restartCount'
```

    Notice there is no more NGINX entry.

14. Take a minute to review what you just learned. It would seem that Kubernetes is a bit like Sigourney Weaver in Alien Resurrection, or perhaps it's a bit more like a synthetic from Blade Runner... in any event, it doesn't matter how many times you knock down a service under Kubernetes control, Kubernetes can easily stand it back up.

Click for Code to Catch Up!

```
kubectl config use-context kubernetes-the-alta3-way
kubectl create -f https://static.alta3.com/projects/k8s/nginx-pod.yaml
sleep 60
kubectl delete -f https://static.alta3.com/projects/k8s/nginx-pod.yaml
kubectl get pods --all-namespaces --sort-by='.status.containerStatuses[0].restartCount'
```

17. Probes

Liveness Probe

 1

Method #1- httpGet

```
apiVersion: v1
kind: Pod
metadata:
  name: sise-lp
spec:
  containers:
  - name: sise
    image: mhausenblas/simpleservice:0.5.0
    ports:
    - containerPort: 9876
    livenessProbe:
      initialDelaySeconds: 2
      periodSeconds: 5
      httpGet:
        path: /health
        port: 9876
      timeoutSeconds: 1
      failureThreshold: 3
```

Method #2- exec command

```
apiVersion: v1
kind: Pod
metadata:
  name: sise-lp
spec:
  containers:
  - name: sise
    image: mhausenblas/simpleservice:0.5.0
    ports:
    - containerPort: 9876
    livenessProbe:
      initialDelaySeconds: 2
      periodSeconds: 5
      exec:
        command:
        - cat
        - /tmp/healthy
      timeoutSeconds: 1
      failureThreshold: 3
```

Readiness Probe

 2

Method #1- httpGet

```
apiVersion: v1
kind: Pod
metadata:
  name: sise-lp
spec:
  containers:
  - name: sise
    image: mhausenblas/simpleservice:0.5.0
    ports:
    - containerPort: 8080
    readinessProbe:
      httpGet:
        path: /health
        port: 8080
      initialDelaySeconds: 3
      periodSeconds: 4
      successThreshold: 2
```

Method #2- exec command

```
apiVersion: v1
kind: Pod
metadata:
  name: sise-lp
spec:
  containers:
  - name: sise
    image: mhausenblas/simpleservice:0.5.0
    ports:
    - containerPort: 8080
    readinessProbe:
      exec:
        command:
        - cat
        - /tmp/healthy
      initialDelaySeconds: 3
      periodSeconds: 4
      successThreshold: 2
```

Distinction between Readiness and Liveness

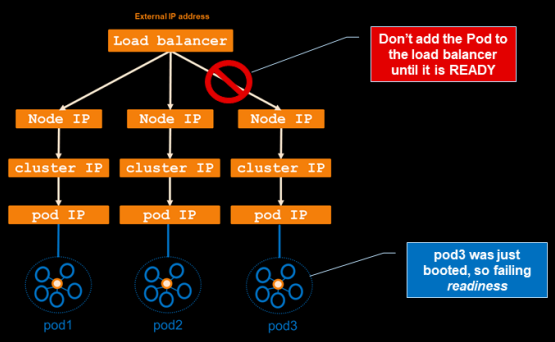
 3

Liveness fail: It's dead!

Readiness fail: It's not ready yet!

Readiness

- Same configuration as a liveness probe
- The Pod is NOT added to the LoadBalancer until it passes the *readiness probe*



Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

18. LivenessProbes and ReadinessProbes

CKAD Objective

- *Understand LivenessProbes and ReadinessProbes*

Lab Objective

This lab will check to see which nodes are ready.

Then this lab will teach how to configure containers to use LivenessProbes and ReadinessProbes and explore their use cases.

If a node isn't ready, there may be a problem with it. Nodes labeled as "ready" can accept pod deployment. However, if a node comes up as "not ready," there's the possibility of a problem... or the node simply hasn't finished initializing.

Similarly, containers can have **ReadinessProbes** inserted into them. This will help services determine if it should start to use the Pod or if it has to wait until all the containers inside of the Pod have become ready.

Also, **LivenessProbes** can be placed into containers as well. These are intended to assure that a container is still performing its essential tasks. If they are deemed not to be alive anymore, even though the container itself is still running, the LivenessProbe will cause a reboot of the container.

Questions for this lab.

What is the difference between liveness and readiness?

Procedure

1. Run setup for this lab

   ```
   student@bchd:~$ setup livenessprobes-and-readinessprobes
   ```

2. Now it's time to work with a **livenessProbe**. This one in particular will cause k8s to send HTTP GET messages every 5 seconds to `http:<POD-IP>:9876/health`. **Create the following manifest** that will make use of our **si**mple**se**rvice image.

   ```
   student@bchd:~$ vim ~/sise-lp.yaml
   ```

3. The following solution has a rather complex `spec.containers`. All available values for `spec.containers` are available at https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/#container-v1-core which may be worth reviewing before you create the following solution:

   Click here to view the contents of **sise-lp.yaml**

   To save and quit out of vim, type: `:wq` and press **Enter**.

4. Create the pod using that manifest, and then port-forward to that pod.

   ```
   student@bchd:~$ kubectl apply -f ~/sise-lp.yaml
   ```

5. Take a look at the "liveness" of your probe. You can do this in a few different ways. You may need to wait a minute while the pod comes up.

   ```
   student@bchd:~$ kubectl describe pods sise-lp | grep -i health

   Liveness:       http-get http://:9876/health delay=2s timeout=1s period=5s #success=1 #failure=3
   ```

   OR

   ```
   student@bchd:~$ kubectl port-forward sise-lp 2224:9876 --address=0.0.0.0
   ```

   AND (in a second tmux pane)

   ```
   student@bchd:~$ curl localhost:2224/health

   {"healthy": true}
   ```

6. Before moving further, review https://github.com/mhausenblas/simpleservice. Pay special attention to the section that reads as follows, 'HEALTH_MIN and HEALTH_MAX ... the min. and max. delay in milliseconds that the /health endpoint responds'

7. Also, take a look at the k8s documentation for liveness and readiness probes. This link has all of the settings available: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/#configure-probes.

8. Create a manifest called, `~/badpod.yaml`

```
student@bchd:~$ vim ~/badpod.yaml
```

9. Create the following manifest. The manifest is named `badpod`, because of the settings we are passing to the environmental variables `HEALTH_MIN` and `HEALTH_MAX`. The values provided within the manifest will cause the service to respond to HTTP GET messages with (at least) a 2 second delay (and up to) a 4 second delay. Note that in this manifest, we are also setting up a `livenessProbe`, to check on the health of the container every 5 seconds, expecting a response to the HTTP GET message within 1 second (smaller than the response time configured within simpleservice). In short, this pod is setup to fail it's LivenessProbes.

Click here to view the contents of **badpod.yaml**

10. Save and exit.

11. Run the following command to create `badpod`.

```
student@bchd:~$ kubectl apply -f ~/badpod.yaml

pod/badpod created
```

12. After 2 seconds Kubernetes will send an HTTP GET to `:9876/health`. When this HTTP GET probe fails 3 consecutive times, Kubernetes will restart the pod. Run the following command every minute to observe this behavior. *Note: Watch the restart count on the pod failing the readiness check.*

```
student@bchd:~$ kubectl get pods
```

13. After observing that restart count is increasing on `badpod`, try looking at `badpod` with a describe command. Again, you're looking for notices that the pod is undergoing excessive restarts. This information should be displayed near the bottom of the display, within the log section.

```
student@bchd:~$ kubectl describe pods badpod
```

14. When you ran the `kubectl describe pods badpod` command, take note that the environmental variables you passed into the container image are present. You're looking for something like the following:

```
Environment:
   HEALTH_MIN:  2000
   HEALTH_MAX:  4000
```

15. Of course, a liveness probe on its own is not enough. We need more probes! **Readiness probes** describe when a container is ready to serve user requests. Containers that fail readiness checks are assumed to be booting and are removed from service load balancers. They are configured similarly to liveness probes.

16. Create a YAML file called, `~/sise-rp.yaml`.

```
student@bchd:~$ vim ~/sise-rp.yaml
```

17. In this manifest, we'll add a section that describes a `readinessProbe`.

Click here to view the contents of **sise-rp.yaml**

18. Save and exit with `:wq`

19. Create the pod we just defined. After 10 seconds, Kubernetes will begin testing readiness of the pod with HTTP GET messages.

```
student@bchd:~$ kubectl apply -f ~/sise-rp.yaml

pod/sise-rp created
```

20. Take a look at the "readiness" of your probe.

```
student@bchd:~$ kubectl describe pods sise-rp | grep -i health

Readiness:      http-get http://:9876/health delay=10s timeout=2s period=3s #success=1 #failure=3
```

> **Performing a Readiness Check**
> Oftentimes when an application first starts up, it isn't ready to handle requests. There is usually some amount of initialization that can take up to several minutes. One thing the service object does is to track which of your pods are ready via a readiness check. Three successive failures of the readiness check will report the pod as "not ready." However, if only one check succeeds, then the pod will again be considered ready.

21. Open up an editor and create a pod manifest. This will be used to create/deploy a pod.

```
student@bchd:~$ vim ~/simpleservice2.yaml
```

22. Create the following manifest, which includes a `readinessProbe`.

Click here to view the contents of **simpleservice2.yaml**

23. Save and exit with `:wq`

24. Create the simpleservice2 pod. The `readinessProbe` will begin running after the pod is started, as the `initialDelaySeconds` is defaulted to 0.

```
student@bchd:~$ kubectl create -f simpleservice2.yaml

pod/simpleservice2 created
```

25. Verify that the pod is running.

```
student@bchd:~$ kubectl get pod simpleservice2

NAME            READY     STATUS    RESTARTS   AGE
simpleservice2   1/1       Running   0          4m
```

26. If your port forward from earlier is still running, cancel it with **ctrl c**.

27. Start port forwarding so that you can browse the pod.

```
student@bchd:~$ kubectl port-forward simpleservice2 2224:9876 --address=0.0.0.0

Forwarding from 0.0.0.0:2224 -> 9876
```

28. You can use *curl* to check the URL for liveness. You will need to open up a new tmux pane in order to do this by doing `Ctrlb` **hands off the keyboard**, `Shift5`. Then curl your pod's liveness:

```
student@bchd:~$ curl -i http://127.0.0.1:2224/health;echo

HTTP/1.1 200 OK
Date: Tue, 14 Apr 2020 22:15:28 GMT
Content-Length: 17
Etag: "b40026a9bea9f5096f4ef55d3d23d6730139ff5e"
Content-Type: application/json
Server: TornadoServer/4.3

{"healthy": true}
```

29. Run `kubectl describe` and confirm the simpleservice2 pod is passing readiness testing. Look at the log displayed at the bottom.

```
student@bchd:~$ kubectl describe pods simpleservice2
```

30. Delete the simpleservice2 pod.

```
student@bchd:~$ kubectl delete pods simpleservice2
```

31. If you'd like, you may try the next two *optional* challenges.

32. **CHALLENGE 01 (OPTIONAL)-** The README page for simpleservice says that there are other paths you could use for liveness checking: `/env`, `/endpoint0`, and `/info`. Edit your YAML to change liveness testing from `/health` to `/info`. Which endpoint you use is left up to you.

33. **CHALLENGE 02 (OPTIONAL)-** Cause the readiness testing to fail. Change the readiness path from `/health` to a non-existing path, `/bummer`. This should cause an on-going series of HTTP 404 failure responses, and therefore, an on-going series of pod restarts, because `/bummer` does not exist as a viable HTTP path inside of the container.

34. All done? Clean up from this lab.

```
student@bchd:~$ kubectl delete pods --all
```

19. Security Context

Overview of Security Context

 1

What?

- Defines operating system security settings (uid, gid, capabilities, SELinux role).
  - **Pod Level:** applies across ALL containers in the Pod.
  - **Container Level:** Specific to one container; overrides settings made at the Pod level.

Why?

- Gives privilege and access control settings for a Pod or Container.
  - Granular permission control.
- Run in privileged or unprivileged mode.
- Make root filesystem read only.
- A good supplement to RBAC and Security Standards. Redundancy in security is a strong best practice.

Pod Security Context

 2

`pod-security-context.yaml`

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: security-context-demo
5  spec:
6    securityContext:
7      runAsUser: 1000
8      runAsGroup: 3000
9      fsGroup: 2000
10   volumes:
11   - name: sec-ctx-vol
12     emptyDir: {}
13   containers:
14   - name: sec-ctx-demo
15     image: busybox
16     command: [ "sh", "-c", "sleep 1h" ]
17     volumeMounts:
18     - name: sec-ctx-vol
19       mountPath: /data/demo
20     securityContext:
21       allowPrivilegeEscalation: false
22       readOnlyRootFilesystem: True
```

Line 2 – the kind is a Pod so this will direct to a Pod security context.

Lines 5/6 – spec.securityContext recognizes the next three lines have user, group, and fsgroup permissions.

Lines 20/21 – sets a securityContext that does not allow for privilege escalation.

Line 22 – sets the root filesystem to read only

How to check your context?

- `kubectl apply`
- `kubectl get pod`
- `kubectl exec` *(into container)*
- `ps` *(check what user processes are running as)*

Container Security Context

 3

`container-security-context.yaml`

```
1  apiVersion: v1
2  kind: Pod
3  metadata:
4    name: security-context-demo-2
5  spec:
6    securityContext:
7      runAsUser: 1000
8    containers:
9    - name: sec-ctx-demo-2
10     image: gcr.io/google-samples/node-hello:1.0
11     securityContext:
12       allowPrivilegeEscalation: false
13       readOnlyRootFilesystem: true
```

Lines 5/6 - spec.securityContext recognizes the next line has user permissions for the Pod.

Lines 11 - 13 - sets a securityContext that does not allow for privilege escalation, and making the root filesystem read only, specifically within the container.

How to check your context?

- `kubectl apply`
- `kubectl get pod`
- `kubectl exec` *(into container)*
- `ps` *(check what user processes are running as)*

20. Understanding Security Contexts for Cluster Access Control

Lab Objective

- Configure a Container and a Pod to use a securityContext.

A Pod or Container may have a SecurityContext applied to it in order to define privileges and access parameters.

It is possible to set all of the containers inside of the Pod to have the same privileges and access parameters by setting the PodSecurityContext. To do this, you can add a securityContext field inside of the Pod specifications. See this truncated example of where to put the securityContext.

```
spec:
  securityContext:
    runAsUser: 1000 # specifies all processes run with user ID 1000
    runAsGroup: 3000 # specifies all processes run with primary group ID 3000
    fsGroup: 2000 # specifies all processes run with secondary group ID 2000
```

Notice how it is at the same level that you would define the containers at. We also will be able to set a securityContext inside of any individual container. That would look like this:

```
containers:
- name: secured-container
  image: mhausenblas/simpleservice
  securityContext:
    runAsUser: 2000
    allowPrivilegeEscalation: false
    readOnlyRootFilesystem: true
```

If you would like to read more about them, check out the Kubernetes Documentation here: <a target="_blank" href="https://kubernetes.io/docs/tasks/configure-pod-container/security-context/> https://kubernetes.io/docs/tasks/configure-pod-container/security-context/

Procedure

1. In Linux, every user has a number assigned to them for tracking purposes (UIDs). The same is true for groups (GIDs). If you wanted to see those assignments, you can (provided you have proper permissions) by displaying the contents of `/etc/passwd`

   ```
   student@bchd:~$ cat /etc/passwd
   ```

2. The first way we are going to secure our pod is to ensure that we are forcing anybody accessing our cluster to run as a specific user, by providing their User ID. Here we are setting the UID to be 1000, as well as disabling the ability to run as a privileged (root) user. Finally, we'll make the root file system read only. Create the following Pod manifest:

   ```
   student@bchd:~$ vim ~/secured_cont.yaml
   ```

   Click here to view the contents of **secured_cont.yaml**

3. Start up your secure Pod.

   ```
   student@bchd:~$ kubectl apply -f ~/secured_cont.yaml

   pod/secured-cont created
   ```

4. Next, verify that the Pod is running.

   ```
   student@bchd:~$ kubectl get pods secured-cont

   NAME           READY   STATUS    RESTARTS   AGE
   secured-cont   1/1     Running   0          85s
   ```

5. Now, let's exec inside of the Container to examine what privileges we have.

   ```
   student@bchd:~$ kubectl exec -it secured-cont -- sh
   ```

6. Once inside of your shell, take a look at the processes you are privy to. The output from this command should show that the USER is 1000, just like we had in our **runAsUser** field of the securityContext.

   ```
   / $ ps
   ```

```
PID   USER     TIME  COMMAND
    1 1000      0:00 sleep 1h
   12 1000      0:00 sh
   17 1000      0:00 ps
```

7. Next, take a look inside of the /data directory. You should only have the `/secured` directory. You will notice that it has the group ID of 2000, which we set while using our fsGroup.

```
/ $ cd /data && ls -l

total 4
drwxrwsrwx    2 root     2000          4096 Feb 18 21:53 secured
```

8. Create a file inside of the /data/secured directory.

```
/data $ echo 'security verified' > secured/security-test.txt
```

9. Inspect the file. This will show you that the user ID is 1000 (set by the **runAsUser** field), and the group ID of 2000 (set by the **fsGroup** field).

```
/data $ ls -l secured

total 4
-rw-r--r--    1 1000     2000            18 Feb 18 21:59 security-test.txt
```

10. To assure that this pod does not allow for any root access, the group ID (gid) needs to not be 0. Although we have set the **fsGroup** to 2000, if we would ignore the **runAsGroup** field, the process still would be able to interact with root group files. That is why we set the **runAsGroup** to 3000. Verify that this is the case by running the following command.

```
/data $ id

uid=1000 gid=3000 groups=2000
```

11. Now, let's try creating a file within our current directory.

```
/data $ touch hello

touch: hello: Read-only file system
```

12. Exit out of the container.

```
/data $ exit
```

13. Now let's get rid of this pod.

```
student@bchd:~$ kubectl delete pod secured-cont

pod "secured-cont" deleted
```

Click for Code to Catch Up!

```
cat /etc/passwd
wget https://labs.alta3.com/courses/kubernetes/files/secured_cont.yaml -O ~/secured_cont.yaml
kubectl apply -f secured_cont.yaml
sleep 60
kubectl get pods secured-cont
kubectl exec secured-cont -- "ps"
kubectl exec secured-cont -- ls -l /data
kubectl exec secured-cont -- sh -c "echo 'security verified' > /data/secured/security-test.txt"
kubectl exec secured-cont -- ls -l /data/secured
kubectl exec secured-cont -- sh -c "cd /data ; id"
kubectl delete pod secured-cont
sleep 20
```

21. Resources

Minimum Required Resources

 1

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mb"
        cpu: "250m"
```

A request

- Amount resources guaranteed to the container
- K8s uses this info to schedule the node
- The container MAY hog far more resources if they are available

Capping Resource Usage

 2

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mb"
        cpu: "250m"
      limits:
        memory: "128Mb"
        cpu: "500m"
```

A limit

- The max resources a container may consume.
- Stops rogue containers from hogging up all the resources on the node

Understanding K8s Resources

 3

```
apiVersion: v1
kind: Pod
metadata:
  name: frontend
spec:
  containers:
  - name: db
    image: mysql
    resources:
      requests:
        memory: "64Mb"
        cpu: "250m"
      limits:
        memory: "128Mb"
        cpu: "500m"
```

A request
- Amount resources guaranteed to the container
- K8s uses this info to schedule the node
- The container MAY hog far more resources if they are available

A limit
- The max resources a container may consume.
- Stops rogue containers from hogging up all the resources on the node

ResourceQuotas

 4

Resource Name	Description
limits.cpu	*Across all Pods in a non-terminal state, the sum of CPU limits cannot exceed this value.*
limits.memory	*Across all Pods in a non-terminal state, the sum of memory limits cannot exceed this value.*
requests.cpu	*Across all Pods in a non-terminal state, the sum of CPU requests cannot exceed this value.*
requests.memory	*Across all Pods in a non-terminal state, the sum of memory requests cannot exceed this value.*

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: dev-rq
spec:
  hard:
    requests.cpu: "10"
    requests.memory: 1Gi
    limits.cpu: "20"
    limits.memory: 3Gi
```

22. Setting an Application's Resource Requirements

CKAD Objective

- *Define an application's resource requirements*

Lab Objective

With Kubernetes, a pod requests the resources required to run its containers. Kubernetes guarantees that these resources are available to the pod. Most common are CPU and memory, but GPU is also available.
We want to create another YAML file in order to accomplish the goal of this lab. This YAML file is actually our pod manifest which will serve to cap resource limits on a pod.

This lab will create a resource request spec for the container.

- Open a new terminal and PUNISH the pod.

Although you have free reign over CPU, memory, and storage resources, as an administrator you understand the need to have the ability to scale. It's a very important part of administering systems.

There is always a balance between what the customers actually need and what they are requesting. You've been through those chats with family and friends. A new product comes out and has X amount of this, Y amount of that, and Z amount of something else. That's the reason they pick this product when, in reality, they may only need half of those resources and honestly only use a quarter of them.

Take the steps ahead of time to limit resources appropriately. If you find more resources are needed later, you can adjust as needed. You'll find this to be an easy process when having to adjust the controls for these limits. Specifically focusing on the website, navigate through the ins and outs of resource requirements.

Questions for this lab.

- Can you raise a resource limit on a live pod?
- Can you reduce the resource limit on a live pod?
- What are the resource limits you can set?

Procedure

1. Run setup for this lab

```
student@bchd:~$ setup setting-an-applications-resource-requirements
```

2. Define the pod with the minimum required resources for where it is allowed to land when deployed. To do this, create a pod manifest.

```
student@bchd:~$ batcat ~/mycode/yaml/linux-pod-r.yaml
```

Click here to view the contents of **linux-pod-r.yaml** 

3. Apply this manifest to the API server in order to create your Pod.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/linux-pod-r.yaml

pod/linux-pod-r created
```

4. Take a look at the resources you have given the Pod.

```
student@bchd:~$ kubectl describe pods linux-pod-r | egrep "cpu|memory"

cpu:        300m
memory:     256Mi
```

> **Capping Resource Usage**
> In addition to setting the resources required by a pod, which establishes the minimum resources available to the pod, a maximum may be set on a pod's resource consumption via resource limits.

5. Now let's add some limits on there as well. You will create a NEW Pod Manifest with the following YAML:

```
student@bchd:~$ batcat ~/mycode/yaml/linux-pod-rl.yaml
```

Click here to view the contents of **linux-pod-rl.yaml**

6. Now create that Pod by submitting the file to the API server.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/linux-pod-rl.yaml

pod/linux-pod-rl created
```

7. Let's use the kubectl `describe` command to find out if our pod has the requested resources and limitations.

```
student@bchd:~$ kubectl describe pods linux-pod-rl | egrep Limits -A 2

Limits:
  cpu:     300m
  memory:  512Mi

student@bchd:~$ kubectl describe pods linux-pod-rl | egrep Requests -A 2

Requests:
  cpu:        300m
  memory:     256Mi
```

8. Update the apt cache inside the Linux pod. First we will need a DNS server set in the config.

```
student@bchd:~$ kubectl exec -it linux-pod-rl -- bash

root@linux-pod-rl:/# echo "nameserver 8.8.8.8" | tee /etc/resolv.conf

nameserver 8.8.8.8

root@linux-pod-rl:/# exit

student@bchd:~$ kubectl exec -it linux-pod-rl -- apt-get update
```

9. Test the limits controls by loading the server and watching CPU and memory utilization. Install htop inside the linux-pod-rl pod.

```
student@bchd:~$ kubectl exec -it linux-pod-rl -- apt-get install htop -y
```

10. Next install `stress` inside the pod.

```
student@bchd:~$ kubectl exec -it linux-pod-rl -- apt-get install stress
```

11. Open a new terminal (or tmux pane) for the next step.

12. Exec into your linux-pod-rl in one terminal (or pane).

```
student@bchd:~$ kubectl exec -it linux-pod-rl -- bash
```

13. In your other terminal (or pane), run htop.

```
student@bchd:~$ kubectl exec -it linux-pod-rl -- htop
```

14. Switch back to your open linux-pod-rl pane with `ctrl` + `b`, take your hands off the keyboard, and then press the direction arrow towards the pane.

15. You might notice that `htop` is showing more memory available than 512 Mb. Inside the container, we're actually seeing a reflection of the total system memory available on the worker node. That's okay, the container is still limited to only using 512 MB.

16. it is time to punish your pod.

```
root@linux-pod-rl:/# stress --vm 1 --vm-bytes 600M --vm-hang 1
```

Your output should look similar to this:

```
stress: info: [426] dispatching hogs: 0 cpu, 0 io, 1 vm, 0 hdd
stress: FAIL: [426] (415) <-- worker 427 got signal 9
stress: WARN: [426] (417) now reaping child worker processes
stress: FAIL: [426] (451) failed run completed in 0s
```

17. Execute the previous command several more times and watch as htop reacts. Htop will show that you exceed the memory limit temporarily before `stress` reports a `FAIL`.

The limits hold the line on utilization! Wow!!!

18. Notice that `spec.containers.resources.limits.memory` is currently set to `"512Mi"`. If you increase this number, more memory will be allowed to be consumed when you `stress` your pod.
19. Exit out of your container (type exit) and stop running the htop (type q).
20. **CHALLENGE 01 (Optional) -** Delete your linux-pod-rl pod, then increase the Memory limits to 1024Mi, then load the pod. You will see that Htop reports greater utilization.

Click for Code to Catch Up!

```
wget https://labs.alta3.com/courses/kubernetes/files/linux-pod-r.yaml -O ~/linux-pod-r.yaml
```

kubectl apply -f mycode/yaml/linux-pod-r.yaml kubectl apply -f mycode/yaml/linux-pod-rl.yaml

23. Monitoring Applications in Kubernetes

Objective

In this lab we'll be setting up a Metrics Server in our new cluster. A Metrics Server is a cluster-wide aggregator of resource usage data. It collects metrics like CPU or memory consumption for containers or nodes, from the Summary API, exposed by Kubelet on each node.

We will confirm success by using `top`.

Procedure

1. Run setup for this lab.

```
student@bchd:~$ setup kubectl-top
```

2. On the bchd machine terminal, run the following command to start up the metrics server inside of your cluster.

```
student@bchd:~$ kubectl apply -f https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
```

3. Wait about 30 seconds. Then, verify that the metrics server service is built and in a *True* state.

```
student@bchd:~$ kubectl get apiservice | grep metrics

v1beta1.metrics.k8s.io                kube-system/metrics-server   True        30s
```

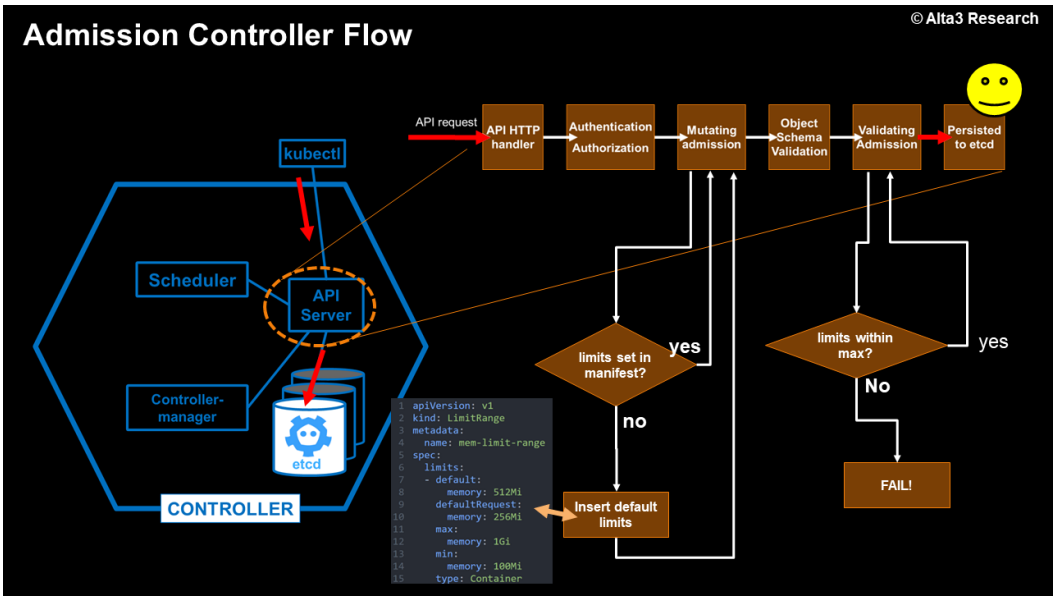
4. Make sure that it is working as expected by issuing *top*ing command. If you get the response `error: metrics not available yet`, that simply means that the metrics server is still gathering the information that it needs. Wait for about 30 to 90 seconds and then try again. **Within a few minutes, it should begin working**

```
student@bchd:~$ kubectl top nodes

NAME     CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
node-1   71m          3%     1765Mi          46%
node-2   80m          4%     1417Mi          37%
```

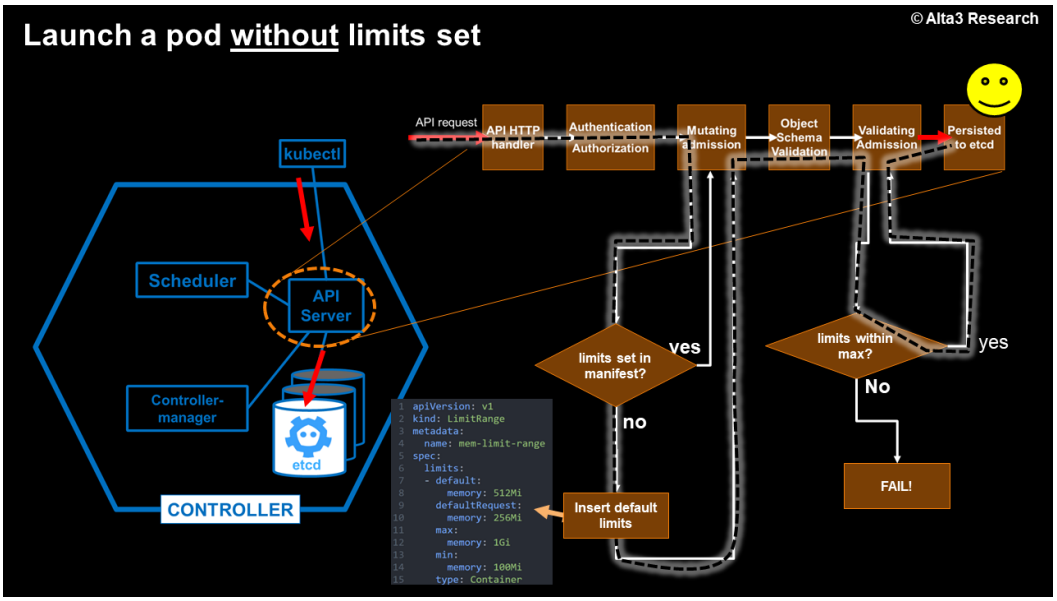
5. Next, top the pods.

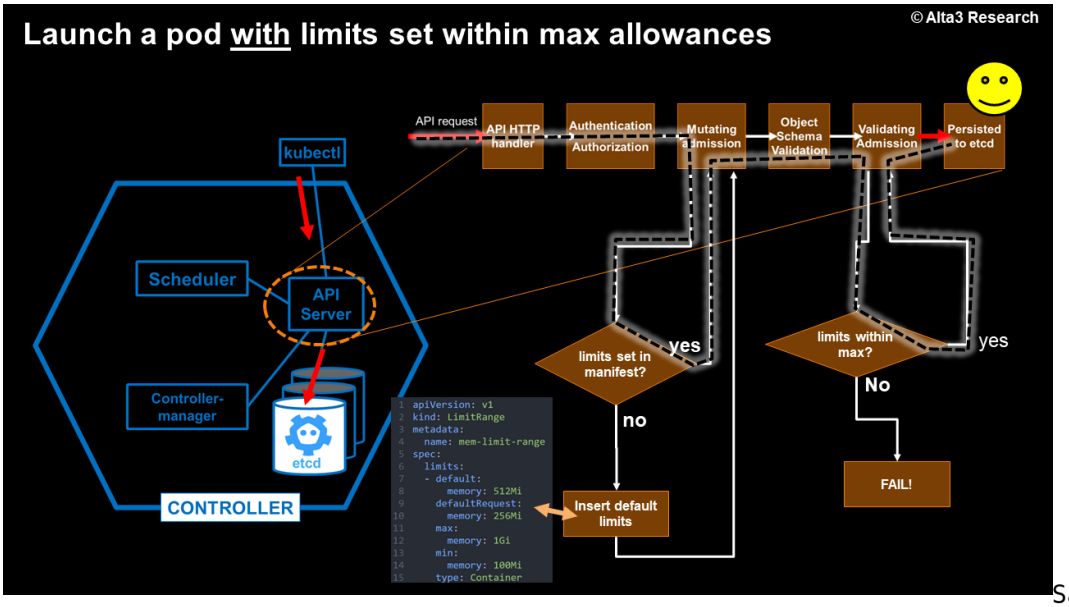
```
student@bchd:~$ kubectl top pods --all-namespaces

NAMESPACE     NAME                                       CPU(cores)   MEMORY(bytes)
kube-system   calico-kube-controllers-659fb845bc-xzbf8   2m           16Mi
kube-system   calico-node-8ct5q                          35m          82Mi
kube-system   calico-node-swgr7                          27m          82Mi
kube-system   kube-dns-688c69f57d-dbxcp                  3m           18Mi
kube-system   metrics-server-847dcc659d-qxcwc            5m           15Mi
```

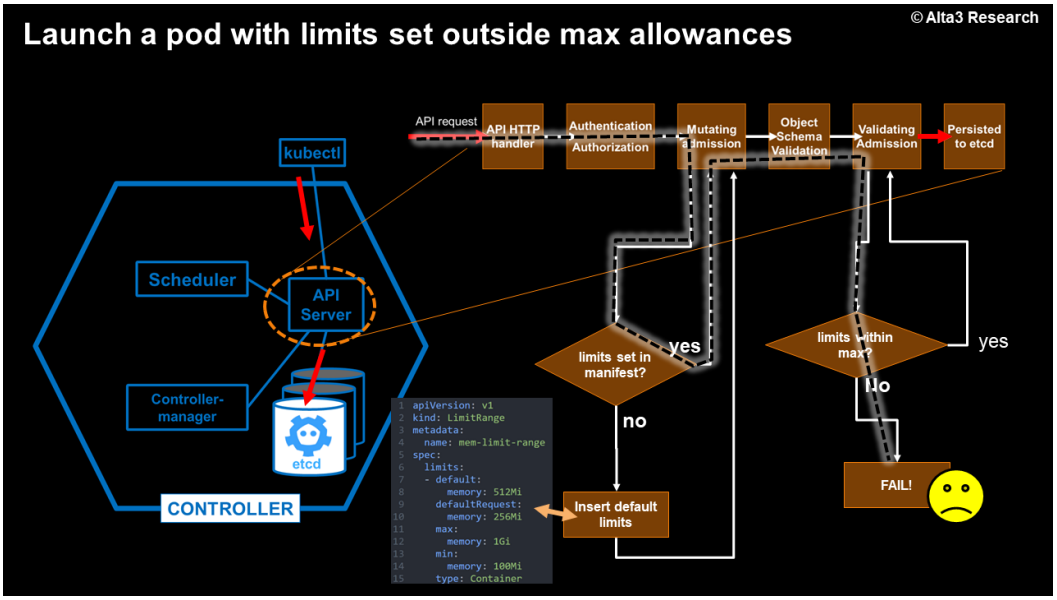
24. Admission Controller Flow

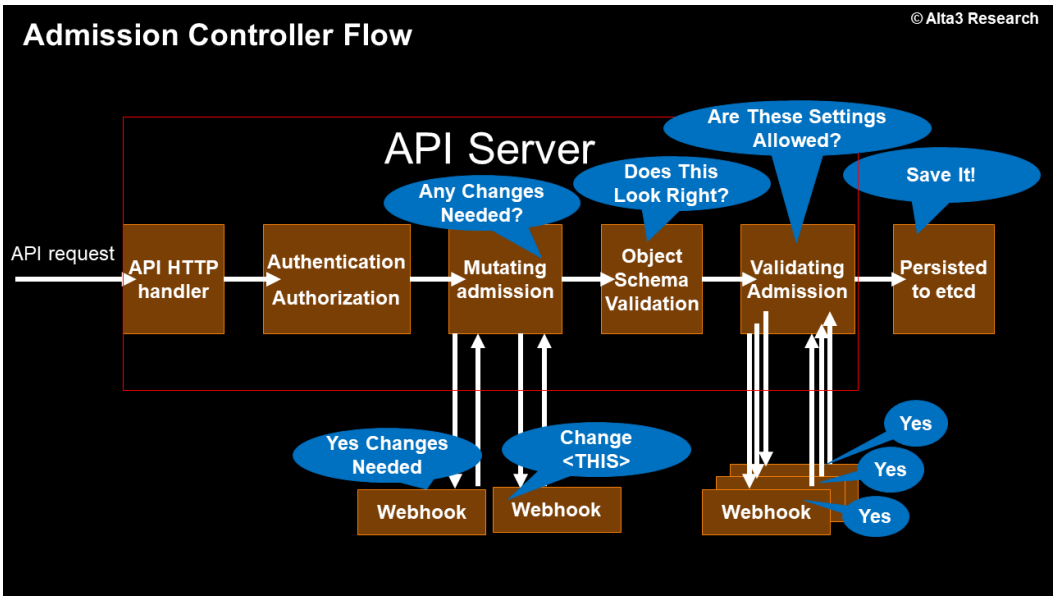


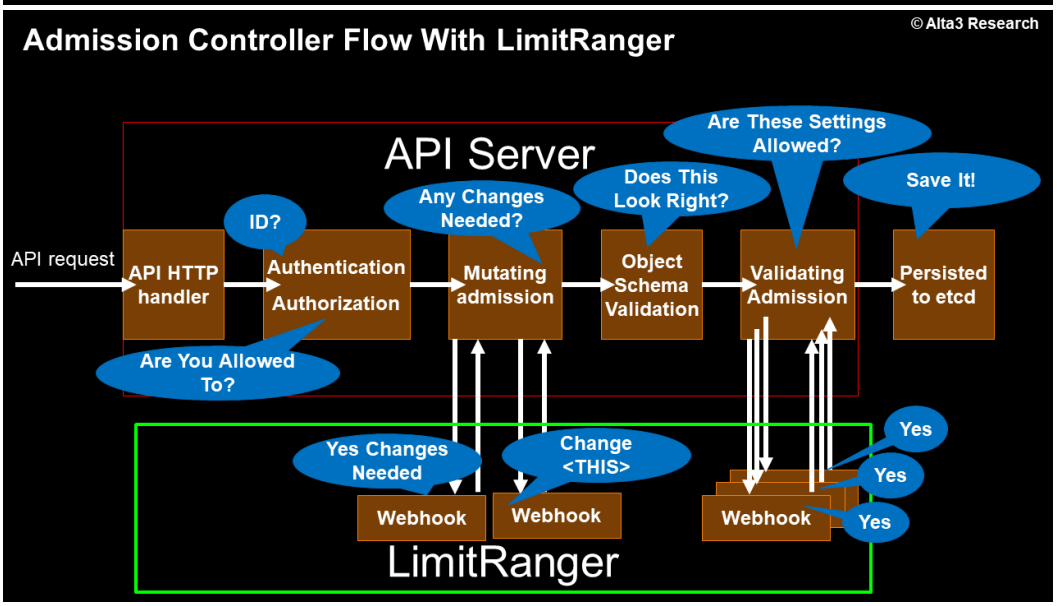
Admission Controller - A piece of code that intercepts requests on their way to the Kubernetes API server after the request is authenticated and authorized. They may either be **mutating** (altering the objects) or **validating** to allow limitations on requests to create, delete, modify or proxy to objects.





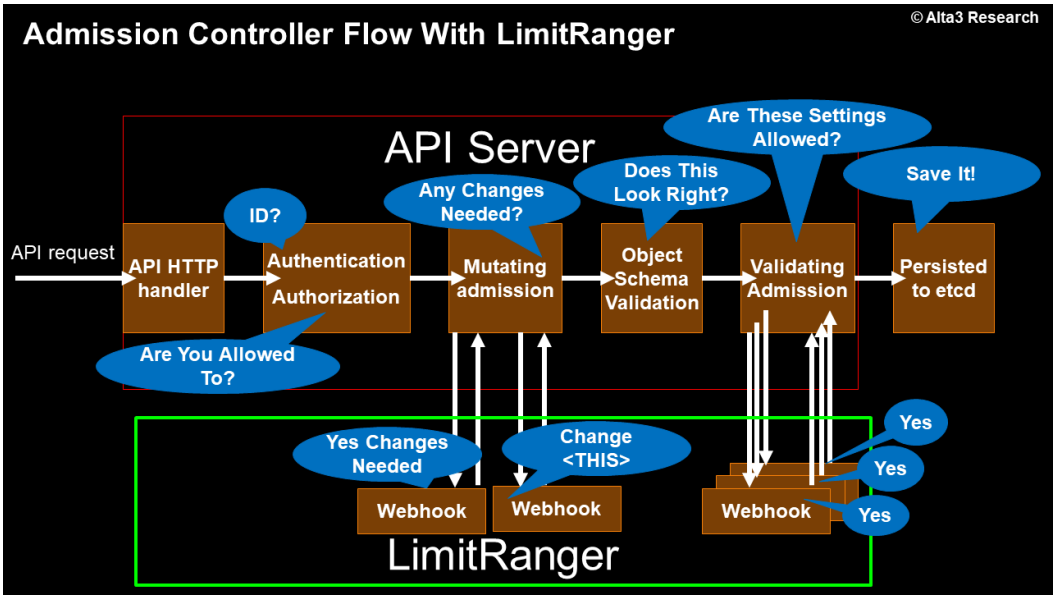






25. Admission Controller - LimitRanger

A **LimitRange** is a Kubernetes object that will allow an administrator to configure the maximum, minimum, and default values that a particular container or pod can have inside of a namespace. It uses the **LimitRanger** controller in order to add the default values in (*mutate*) or to verify that the values provided fall within the minimum and maximum ranges (*validate*).



Procedure - Admission Controller

1. Run this script.

```
student@bchd:~$ setup admission-controller
```

2. First, let's create a Pod that does not use a LimitRange

```
student@bchd:~$ kubectl run no-lr --image=nginx:1.19.6
```

3. Let's inspect our Pod to see if there are any requests placed on our Pod.

```
student@bchd:~$ kubectl get pod no-lr -o yaml | grep request -A 5
```

> There should **not** be anything returned right now

4. Now let's create the following **LimitRange** resource manifest. First let's take a look at it.

```
student@bchd:~$ batcat ~/mycode/yaml/lim-ran.yml
```

Click here to view the contents of **lim-ran.yml**

5. Let's instantiate the LimitRange now.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/lim-ran.yml
```

6. Next, let's create a Pod that will use the LimitRange.

```
student@bchd:~$ kubectl run lr --image=nginx:1.19.6
```

7. And let's see that the new pod has been mutated to include having requests via the LimitRanger.

```
student@bchd:~$ kubectl get pod lr -o yaml | grep request -A 5
```

```
    kubernetes.io/limit-ranger: 'LimitRanger plugin set: memory request for container
      lr; memory limit for container lr'
    creationTimestamp: "2021-10-20T14:13:25Z"
    labels:
    run: lr
    managedFields:
    --
      requests:
        memory: 256Mi
    terminationMessagePath: /dev/termination-log
    terminationMessagePolicy: File
    volumeMounts:
    - mountPath: /var/run/secrets/kubernetes.io/serviceaccount
```

Congratulations! We have successfully used the LimitRanger Admission Controller via a Mutating Webhook to change the information added to etcd about this Pod. Specifically, we have used it to add the default amount of requests to be 256Mi to our container.

Be sure to delete the LimitRanger object! Otherwise having it active may cause problems in future labs.

```
student@bchd:~$ kubectl delete -f ~/mycode/yaml/lim-ran.yml
```

User Authentication and Authorization

1 ©Alta3 Research

Two Categories of Users:

Normal User

Represents a human user who must be authenticated

- Cannot be added via an API call
- Any user who presents a valid certificate signed by the cluster's certificate authority (CA) is considered authenticated.

Service Account

Provides an identity for processes that run in a pod

- Managed by the Kubernetes API
- Bound to specific namespaces
- Created automatically by the API server, or manually through API calls
- Tied to a set of credentials stored as Secrets, which are mounted into pods allowing in-cluster processes to communicate with the Kubernetes API.

Roles and ClusterRole

2 ©Alta3 Research

Role Based Access Control (RBAC):

Contains rules that represent a set of permissions.

Role

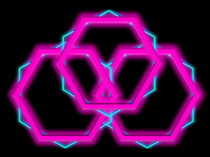
Sets permissions within a particular namespace

- When you create a Role, you must specify the namespace it belongs in.

ClusterRole

Defines permissions for a role that is cluster-wide.

- A non-namespaced resource.
- Define permissions to be granted within individual namespace(s).
- Define permissions to be granted across all namespaces.
- Define permissions on cluster-scoped resources (i.e. nodes)



If you want to define a role within a namespace, use a Role.
If you want to define a role that is cluster-wide, use a ClusterRole.

Important: Permissions are purely additive (there are no "deny" rules).

RoleBinding and ClusterRoleBinding

3 ©Alta3 Research

Binding Roles to Users
Grants permissions defined in a role to a user or set of users.

RoleBinding
Grants permissions within a specific namespace.
- Holds a list of subjects (users, groups, or service accounts) and a reference to the 'Role' or 'ClusterRole' being granted.
- A RoleBinding may reference any Role in the same namespace.
- Can reference a ClusterRole to the namespace of the RoleBinding.

ClusterRoleBinding
Grants permissions cluster-wide.
- If you want to bind a ClusterRole to all the namespaces in your Cluster, use a ClusterRoleBinding.

Important: The name of a RoleBinding or ClusterRoleBinding object must be A valid path segment name. The name may not be "." or ".." and the name may not contain "/" or "%".

Normal User Onboarding

4 ©Alta3 Research

Role

1
```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  namespace: production
  name: t3-support
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "watch", "list"]
```

CertificateSigningRequest

2
```
apiVersion: certificates.k8s.io/v1
kind: CertificateSigningRequest
metadata:
  name: alice
spec:
  username: alice
  signerName: kubernetes.io/kube-apiserver-client
  groups:
  - system:authenticated
  usages:
  - client auth
  request: KEY
```

RoleBinding

3
```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding # Case Sensitive
metadata:
  name: t3-support
  namespace: production
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: t3-support
  apiGroup: rbac.authorization.k8s.io
```

Procedure
1. Create a Role granting permissions for a user
2. Create a Certificate and add it as a resource
3. Bind the Certificate and the Role together

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

26. Roles

Objective

The RBAC API declares four kinds of Kubernetes objects: Role, ClusterRole, RoleBinding, and ClusterRoleBindings. Let's begin with Roles, which allows for namespaced specific permissions. As a reminder, Kubernetes RBAC is purely additive (there are no deny rules). A Role ALWAYS sets permissions within a particular namespace, so when you create a role, you have to specify a namespace. Let's break it down. Be sure to review Kubernetes documentation for a full breakdown as to how this works: https://kubernetes.io/docs/reference/access-authn-authz/rbac/#api-overview.

Simply to serve as a reminder, the following namespaces have been made in earlier labs:

Namespace
test
prod
dev

Procedure - Authentication and Authorization

1. Run the setup for this lab.

```
student@bchd:~$ setup RBAC-authentication-authorization-lab
```

2. Verify the Test, Prod, and Dev namespaces are created:

```
student@bchd:~$ kubectl get ns

NAME              STATUS   AGE
default           Active   2d19h
dev               Active   39m
kube-node-lease   Active   2d19h
kube-public       Active   2d19h
kube-system       Active   2d19h
prod              Active   39m
test              Active   39m
```

> Remember - Roles are 'Namespaced,' meaning that when you define a role you MUST define the namespace it is attached to.

3. There is a developer who needs "view only" permissions on the Production namespace to provide Tier 3 Tech support. Let's create this role, and name it *t3-support*.

```
student@bchd:~$ vim t3-support.yaml
```

Click here to view the contents of **t3-support.yaml**

> Write the file and quit vim: `wq!`

4. Let's go ahead and create the role in our Kubernetes Cluster.

```
student@bchd:~$ kubectl apply -f t3-support.yaml

role.rbac.authorization.k8s.io/t3-support created
```

5. Next, we're going to have to create a certificate for our developer, Alice. Remember, there is no explicit user account resource to add for users. Instead, anyone who holds a valid cert will be able to authenticate and be granted permissions within the Cluster. We'll need to generate an rsa key for Alice in order to Authenticate them.

```
student@bchd:~$ openssl genrsa -out alice.key 2048
```

6. Next we can use our RSA key to generate a new certificate signing request. Note that you must use the group name of system:basic-user as well as the Common Name (CN) of the user's name, Alice.

```
student@bchd:~$ openssl req -new -key alice.key -out alice.csr -subj "/C=US/ST=PA/L=Harrisburg/O=system:basic-user/OU=system:basic-user/CN=alice/emailAddress=support@alta3.com"
```

7. Confirm the details of our CSR.

```
student@bchd:~$ openssl req -text -noout -verify -in alice.csr

...
Subject: C = US, ST = PA, L = Harrisburg, O = system:basic-user, OU = system:basic-user, CN = alice, emailAddress = support@alta3.com
...
```

8. Next, we need to convert the signing request to base64 to read it into Kubernetes. Run the following command.

```
student@bchd:~$ cat alice.csr | base64 | tr -d "\n" > alicekey
```

9. Let's take a look at the key we just made.

```
student@bchd:~$ cat alicekey && echo

LS0tLS1CRUdJTiBDRVJUSUZJQ0FURSBSRVFVRVNULS0tLS0KTUlJQz ...
```

10. Next, we need to setup a Certificate Signing Request in order to Authorize Alice. This is done through a manifest which we'll need to create.

```
student@bchd:~$ vim alicetemplate.yaml
```

Click here to view the contents of **alicetemplate.yaml**

> Write the file and quit vim: `wq!`

11. Naturally, we can't leave ***request*** with the value ***key*** - but would instead insert the key we made earlier into the file. So, we're going to put the Linux Stream Editor (sed) to good use!

```
student@bchd:~$

sed "s/KEY/`cat alicekey`/" alicetemplate.yaml > alice-csr.yaml
```

> The sed command below is formatted as follows: Subsitute `s` all occurences of KEY `KEY` with the contents of alicekey `alicekey` within the file alicetemplate.yaml `alicetemplate.yaml`. Send the STDOUT > to the file alice-csr.yaml `alice-csr.yaml`.

12. Let's apply the Certificate Signing Request to our cluster!

```
student@bchd:~$ kubectl apply -f alice-csr.yaml

certificatesigningrequest.certificates.k8s.io/alice created
```

13. As with any resource, we can use ***kubectl get*** to retrieve information about our brand new csr!

```
student@bchd:~$ kubectl get csr

NAME    AGE   SIGNERNAME                            REQUESTOR   REQUESTEDDURATION   CONDITION
alice   97s   kubernetes.io/kube-apiserver-client   admin       <none>              Pending
```

14. However, this does not complete the process. This was a 'Certificate Signing Request.' As the Cluster Administrator, we will need to approve the certificate.

```
student@bchd:~$ kubectl certificate approve alice

certificatesigningrequest.certificates.k8s.io/alice approved
```

15. With our User, Role, and Certificates in place, it is now time to bind it all together! For this, we will need to create a RoleBinding.

```
student@bchd:~$ vim t3-support-binding.yaml
```

Click here to view the contents of **t3-support-binding.yaml**

> Write the file and quit vim: `wq!`

16. Now, we'll apply the RoleBinding to put everything in place!

```
student@bchd:~$ kubectl apply -f t3-support-binding.yaml

rolebinding.rbac.authorization.k8s.io/management created
```

17. With all of this in place, we should now be able to test whether or not Alice can perform the functions we defined in the Role we provisioned at the beginning of this lab. We can do this with the ***kubectl auth can-i*** command.

```
student@bchd:~$ kubectl auth can-i get pods -n prod --as alice
```

```
yes
```

> The ***kubectl auth can-i*** command is formatted as follow: The command `kubectl auth can-i` followed by the verb `get` then the resource `pods` in the production namespace `-n production` as alice `--as alice`

18. CHALLENGE! Perform the ***kubectl auth can-i*** command to determine if Alice can do the following:
    - watch pods on production
    - list pods on production
    - get secrets
    - get pods on testing

> Record your answers, ready to inform the instructor of your findings.

We've now setup Alice to Authenticate and Authorize T3 Support on Production!

27. Distributing Access

Objective

Now that we've created the necessary certificate, role, and role binding - we have to provide these to our Developer providing T3 Support to our Production Namespace. There are two key components required of a user to gain access to kubectl commands within the cluster - a certificate, and a Key. As a Cluster Administrator, it will be up to you to disseminate the information from the cluster to the appropriate persons in ways that you/your company find secure. This procedure will show you how to retrieve the information needed to give to users, and then introduce you to the steps needed for them to set up their own context.

As a reminder, the following namespaces have been made in earlier labs:

Namespace
test
prod
dev

Note: THIS MUST BE PERFORMED WITHIN 1 HOUR OF THE PREVIOUS LAB.

There is a garbage collection process that automatically deletes Approved CSRs after 1 hour. Read All About It!.

In case you went past the 1 hour time limit...
1. Generate the CSR again.

```
student@bchd:~/ kubectl apply -f alice-csr.yaml
```

1. Approve the CSR again.

```
student@bchd:~/ kubectl certificate approve alice
```

Procedure - Distributing Access to Users

1. First, let's take a look at the decoded version of your csr.

```
student@bchd:~$ kubectl get csr alice -o jsonpath='{.status.certificate}' | base64 --decode

-----BEGIN CERTIFICATE-----
MIIDuDCCAqCgAwIBAgIRALrQhmvAEewTl6mfuOv7fFgwDQYJKoZIhvcNAQELBQAw
cDELMAkGA1UEBhMCVVMxFTATBgNVBAgTDFBlbm5zeWx2YW5pYTETMBEGA1UEBxMK
SGFycmlzYnVyZzETMBEGA1UEChMKS3ViZXJuZXRlczELMAkGA1UECxMCQ0ExEzAR
BgNVBAMTCkt1YmVybmV0ZXMwHhcNMjIwNTE3MjMzMzA1WhcNMjMwNTE3MjMzMzA1
WjB3MQswCQYDVQQGEwJVUzELMAkGA1UECBMCUEExEzARBgNVBAcTCkhhcnJpc2J1
cmcxGjAYBgNVBAoTEXN5c3RlbTpiYXNpYy11c2VyMRowGAYDVQQLExFzeXN0ZW06
YmFzaWMtdXNlcjEOMAwGA1UEAxMFYWxpY2UwggEiMA0GCSqGSIb3DQEBAQUAA4IB
DwAwggEKAoIBAQDXJQ0ENxu+L08XHj9FwO2ifhIxw+QSluM9PPAXqoJMveNUi441
jUKQYdctQ9z6gNCR8vMUYq4UOg4Rk3stBGpZVqJnFaMTIxCuujLHuB2I+pusVdcH
86OosifQtuzt+IVEhs1hIHmsf+e2wDa1/6aqKkAocS4qjv1xHHy/gZIFMFBVhWH/
2GGvkYwdEnjFchCYHiF7Vp0JViTdCb/XvSc3mJ/dlp7CV2SE3UmjD/nZ7hmjPagb
E8VWSmpv8XPVBxBqhiczzrk/MOj6gMCijcctMfh2H4ISVTAP659SUPy5BCK88gm/
Cwf53CLtcAjJZEwfJnhGjHws4a2g52bdVLw7AgMBAAGjRjBEMBMGA1UdJQQMMAoG
CCsGAQUFBwMCMAwGA1UdEwEB/wQCMAAwHwYDVR0jBBgwFoAUlykafY04SyVNYDS7
17xjgq+WqagwDQYJKoZIhvcNAQELBQADggEBAGQBN4lBIPyiGnoA7NqO4Hv2CfM9
6RFWhWsu1ZYdGgJcWYZjPl3Nmk1ShvUuShluuf/leL75/PUd9xZQ7ma+tEVha548
h/L7lAJKRhftUDkX/VBs5qpEigfoPjXGbpR0lbhWyfcO/TMa6egKN72CgGJtrfvx
sCftX7BtVqK8NdhnwQYx35VSDgLTgsuQzq9yT07hcbyQktUYuuJC+CVcFR3vq9E8
MQolpTl3YxK2eQNkODU+t9cORGQpe3W9KOgGzzk46phb58d9iGw0qzK17h+oed3u
qcBzp/d9FAgqt9e57x1tfU/aSGBXUnyn0FkRAbh0JaYuTRjmRvBvaEpIaCw=
-----END CERTIFICATE-----
```

2. Let's redirect that output to a file called **alice.crt**.

```
student@bchd:~$ kubectl get csr alice -o jsonpath='{.status.certificate}' | base64 --decode > alice.crt
```

3. At this point, the Administrator has to select how to share the secret information to their users. To configure their **.kube/config** file, our **alice** user would need these files:
    - **alice.crt**
    - **alice.key**
4. To familiarize ourselves with **.kube/config** let's go ahead and take a look:

```
student@bchd:~$ batcat .kube/config

19   | users:
20   | - name: admin
21   |   user:
22   |     client-certificate: /home/student/k8s-certs/admin.pem
23   |     client-key: /home/student/k8s-certs/admin-key.pem
```

> Do not manually edit the .kube/config file. Kubernetes has a built in configuration tool to do this for us.

5. Traditionally, we would perform these next few steps on the target user's workstation. However, for our learning purposes, we'll perform them on our own workstation, and create a context as though we ourselves were Alice. Let's add the alice user to our .kube/config file to verify they would have access.

```
student@bchd:~$ kubectl config set-credentials alice --client-key=alice.key --client-certificate=alice.crt

User "alice" set
```

6. Next, we'll create a context in order to access the cluster as the alice user.

```
student@bchd:~$ kubectl config set-context alice --cluster=kubernetes-the-alta3-way --user=alice

Context "alice" created
```

7. Now we can switch over to the alice context.

```
student@bchd:~$ kubectl config use-context alice

Switched to context "alice".
```

8. And now we are accessing our cluster as Alice would. We can try out commands we know should work as well as ones we know should fail.

```
student@bchd:~$ kubectl get pods
```

> This should fail.

```
Error from server (Forbidden): pods is forbidden: User "alice" cannot list resource "pods" in API group "" in the namespace "default"
```

9. Now let's verify Alice can run get pods on production.

```
student@bchd:~$ kubectl get pods -n prod
```

> This should succeed, showing that there are no resources on the target namespace.

10. Alice can't keep secrets (or view them), so this should fail.

```
student@bchd:~$ kubectl get secrets

Error from server (Forbidden): secrets is forbidden: User "alice" cannot list resource "secrets" in API group "" in the namespace "default"
```

11. This also should fail because alice is only allowed in the production namespace.

```
student@bchd:~$ kubectl get pods -n test

Error from server (Forbidden): pods is forbidden: User "alice" cannot list resource "pods" in API group "" in the namespace "test"
```

12. Now, let's get back to being the cluster admin again.

```
student@bchd:~$ kubectl config use-context kubernetes-the-alta3-way

Switched to context "kubernetes-the-alta3-way".
```

Kubernetes Logging

kubectl log flags

 1

Name	Shorthand	Default	Usage
all-containers		false	Get all containers' logs in the Pod(s)
container	c		Prints the logs of a specified container
follow	f	false	Specify if the logs should be streamed
since		0s	Only return logs new than relative duration
tail		-1	Lines of recent log files to display
timestamps		false	Include timestamps on each line of log output

…and more!

28. Utilize Container Logs

Lab Objective

This lab takes a look at some of the more advanced techniques of logging with and for Kubernetes. This includes basic logging with pods and containers, using a sidecar pod with Fluentd, as well as centralized logging with best practice of separation from the cluster.

Procedure

Basic Logging

1. In this example, we take a look at a pod specification that has two containers writing text to standard out.

```
student@bchd:~$ vim ~/counter-pod.yaml
```

Click here to view the contents of **counter-pod.yaml**

2. While simple on the outside, Kubernetes inherent logging can show you a great deal more than just that. Below is a list of flags for the kubectl logs command. We will utilize some of these going forward.

Name	Shorthand	Default	Usage
all-containers		false	Get all containers' logs in the pod(s)
container	c		Prints the logs of a specified container
follow	f	false	Specify if the logs should be streamed
since		0s	Only return logs new than relative duration
tail		-1	Lines of recent log files to display
timestamps		false	Include timestamps on each line of log output

1. Now that we have the file created, run the pod.

```
student@bchd:~$ kubectl apply -f ~/counter-pod.yaml

pod/counter created
```

2. Verify that the pod has been created.

```
student@bchd:~$ kubectl get pods counter

NAME      READY   STATUS    RESTARTS   AGE
counter   2/2     Running   0          13s
```

3. Let's check what the output is in the logs. Run the following command to try to look at the logs of the Pod we just created.

```
student@bchd:~$ kubectl logs counter

Defaulted container "count" out of: count, countby3
0: Wed Sep 21 16:08:25 UTC 2022
1: Wed Sep 21 16:08:26 UTC 2022
2: Wed Sep 21 16:08:27 UTC 2022
3: Wed Sep 21 16:08:28 UTC 2022
4: Wed Sep 21 16:08:29 UTC 2022
5: Wed Sep 21 16:08:30 UTC 2022
6: Wed Sep 21 16:08:31 UTC 2022
```

> This command used to error in versions earlier than Kubernetes v1.24 -- but now, the **kubectl logs** command will default to the first container specified in the manifest and give the log for that container, while informing the user of this decision.

4. When you have a Pod with more than one container inside of it, you **can** specify which container you wish to view the logs of. Let's try that again with our **count** container.

```
student@bchd:~$ kubectl logs counter -c count
```

```
0: Mon Nov 15 14:57:39 UTC 2021
1: Mon Nov 15 14:57:40 UTC 2021
2: Mon Nov 15 14:57:41 UTC 2021
3: Mon Nov 15 14:57:42 UTC 2021
4: Mon Nov 15 14:57:43 UTC 2021
5: Mon Nov 15 14:57:44 UTC 2021
6: Mon Nov 15 14:57:45 UTC 2021
7: Mon Nov 15 14:57:46 UTC 2021
```

5. Great! Now let's try accessing the logs from the **countby3** container.

```
student@bchd:~$ kubectl logs counter -c countby3

Mon Nov 15 15:02:55 UTC 2021: 105
Mon Nov 15 15:02:58 UTC 2021: 106
Mon Nov 15 15:03:01 UTC 2021: 107
Mon Nov 15 15:03:04 UTC 2021: 108
Mon Nov 15 15:03:07 UTC 2021: 109
Mon Nov 15 15:03:10 UTC 2021: 110
Mon Nov 15 15:03:13 UTC 2021: 111
Mon Nov 15 15:03:16 UTC 2021: 112
```

6. Now, let's try to follow the logs of the **count** container.

```
student@bchd:~$ kubectl logs counter -c count -f

444: Mon Nov 15 15:05:04 UTC 2021
445: Mon Nov 15 15:05:05 UTC 2021
446: Mon Nov 15 15:05:06 UTC 2021
447: Mon Nov 15 15:05:07 UTC 2021
448: Mon Nov 15 15:05:08 UTC 2021
449: Mon Nov 15 15:05:09 UTC 2021
450: Mon Nov 15 15:05:10 UTC 2021
451: Mon Nov 15 15:05:11 UTC 2021
```

> To exit the logs, do a `Ctrlc` since you used the option of **-f** to "follow" them.

7. Mini Challenge: Follow the logs of the **countby3** container.

> You got this!

8. Now let's try to get the logs from all of our containers in this Pod.

```
student@bchd:~$ kubectl logs counter --all-containers

Mon Nov 15 15:06:46 UTC 2021: 182
Mon Nov 15 15:06:49 UTC 2021: 183
Mon Nov 15 15:06:52 UTC 2021: 184
Mon Nov 15 15:06:55 UTC 2021: 185
Mon Nov 15 15:06:58 UTC 2021: 186
Mon Nov 15 15:07:01 UTC 2021: 187
Mon Nov 15 15:07:04 UTC 2021: 188
Mon Nov 15 15:07:07 UTC 2021: 189
```

> Wait, this looks like we just got the logs from the **countby3** container! That is because the logs are read from the first container first, and the second container second, not mixed together. If you look back through the history of the output or use a tool like **less**, you can view all of them.

9. Another way to look at some of the logs from all of our containers is to specify a **since** flag. Let's see what has happened in both of our containers for the last 10 seconds.

```
student@bchd:~$ kubectl logs counter --all-containers --since 10s
```

```
799: Mon Nov 15 15:10:59 UTC 2021
800: Mon Nov 15 15:11:00 UTC 2021
801: Mon Nov 15 15:11:01 UTC 2021
802: Mon Nov 15 15:11:02 UTC 2021
803: Mon Nov 15 15:11:03 UTC 2021
804: Mon Nov 15 15:11:04 UTC 2021
805: Mon Nov 15 15:11:05 UTC 2021
806: Mon Nov 15 15:11:06 UTC 2021
807: Mon Nov 15 15:11:07 UTC 2021
808: Mon Nov 15 15:11:08 UTC 2021
Mon Nov 15 15:11:01 UTC 2021: 267
Mon Nov 15 15:11:04 UTC 2021: 268
Mon Nov 15 15:11:07 UTC 2021: 269
```

10. Or if we want to just look at the last 5 lines of logs for all of our containers, we can use the **tail** flag.

```
student@bchd:~$ kubectl logs counter --all-containers --tail 5

932: Mon Nov 15 15:13:13 UTC 2021
933: Mon Nov 15 15:13:14 UTC 2021
934: Mon Nov 15 15:13:15 UTC 2021
935: Mon Nov 15 15:13:16 UTC 2021
936: Mon Nov 15 15:13:17 UTC 2021
Mon Nov 15 15:13:04 UTC 2021: 308
Mon Nov 15 15:13:07 UTC 2021: 309
Mon Nov 15 15:13:10 UTC 2021: 310
Mon Nov 15 15:13:13 UTC 2021: 311
Mon Nov 15 15:13:16 UTC 2021: 312
```

11. Although our logs already have timestamps on them, let's also see how the kubectl logs flag of **timestamps** is able to apply their own.

```
student@bchd:~$ kubectl logs counter --all-containers --tail 5 --timestamps

2021-11-15T15:16:50.488034356Z 1149: Mon Nov 15 15:16:50 UTC 2021
2021-11-15T15:16:51.490242494Z 1150: Mon Nov 15 15:16:51 UTC 2021
2021-11-15T15:16:52.491539522Z 1151: Mon Nov 15 15:16:52 UTC 2021
2021-11-15T15:16:53.492712346Z 1152: Mon Nov 15 15:16:53 UTC 2021
2021-11-15T15:16:54.494665484Z 1153: Mon Nov 15 15:16:54 UTC 2021
2021-11-15T15:16:41.067750117Z Mon Nov 15 15:16:41 UTC 2021: 380
2021-11-15T15:16:44.069552746Z Mon Nov 15 15:16:44 UTC 2021: 381
2021-11-15T15:16:47.071627156Z Mon Nov 15 15:16:47 UTC 2021: 382
2021-11-15T15:16:50.072944094Z Mon Nov 15 15:16:50 UTC 2021: 383
2021-11-15T15:16:53.074621152Z Mon Nov 15 15:16:53 UTC 2021: 384
```

12. **Great Job!** You have just tried out all of the various flags that kubectl logs allows you to use. Now take a few minutes and feel free to try grabbing the logs from them as well.

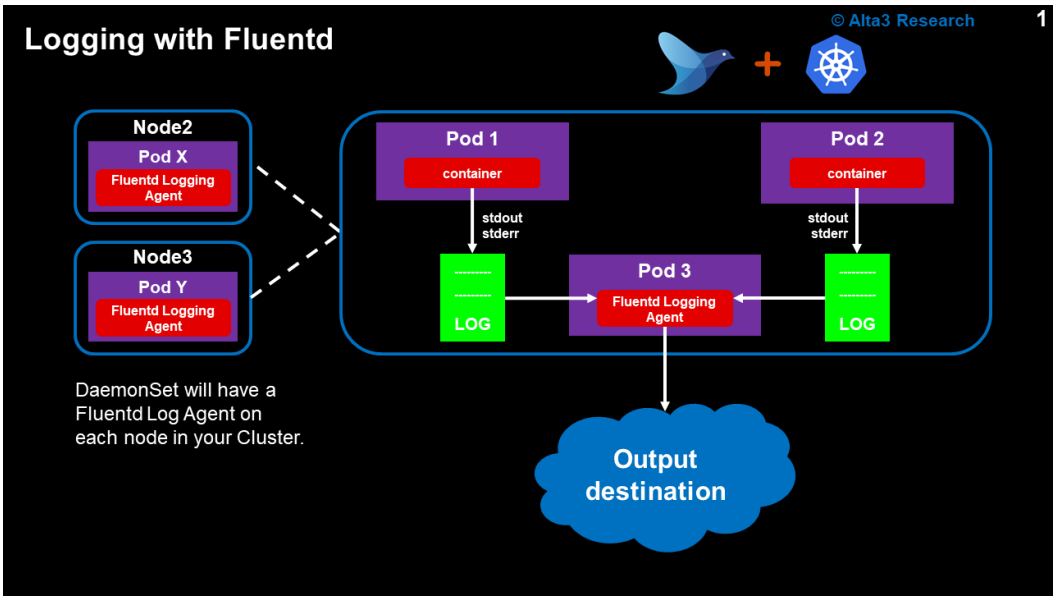
13. Before moving on to our next topic, please remove this logging Pod.

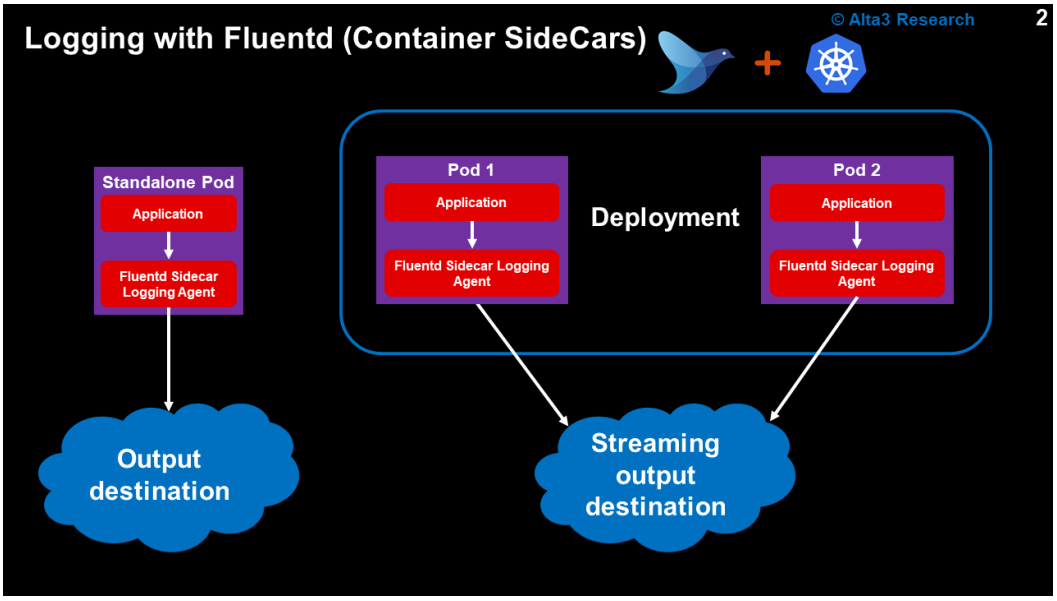
```
student@bchd:~$ kubectl delete -f ~/counter-pod.yaml
```

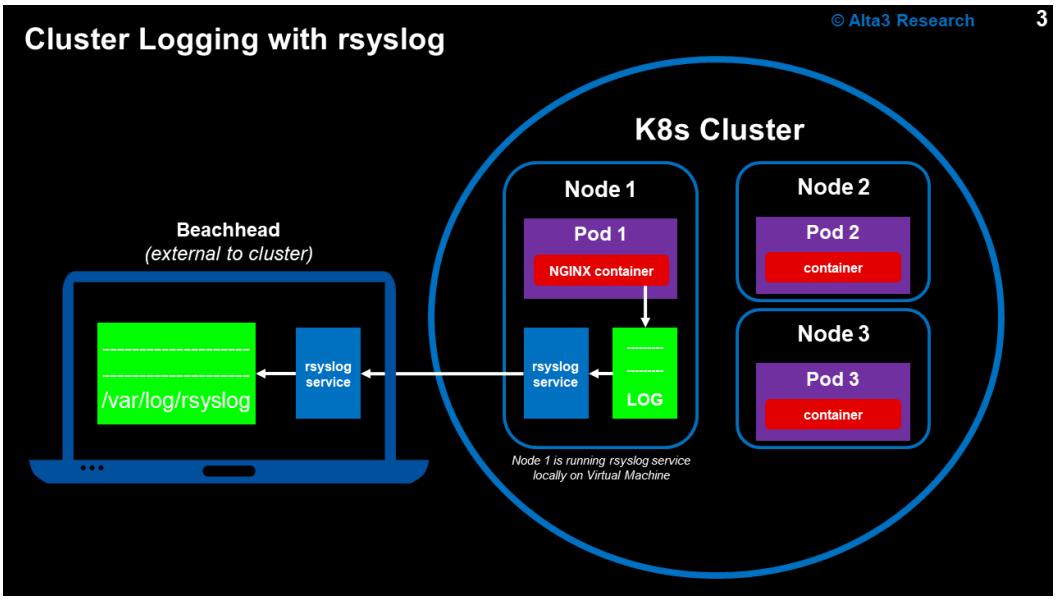
Click for Code to Catch Up!

```
wget https://labs.alta3.com/courses/kubernetes/files/counter-pod.yaml -O ~/counter-pod.yaml
kubectl apply -f ~/counter-pod.yaml
sleep 10
kubectl get pods counter
kubectl logs counter
kubectl logs counter -c count
kubectl logs counter -c countby3
kubectl logs counter -c count -f > /dev/null 2>&1 &
kubectl logs counter --all-containers
kubectl logs counter --all-containers --since 10s
kubectl logs counter --all-containers --tail 5
kubectl logs counter --all-containers --tail 5 --timestamps
kubectl delete -f ~/counter-pod.yaml
```

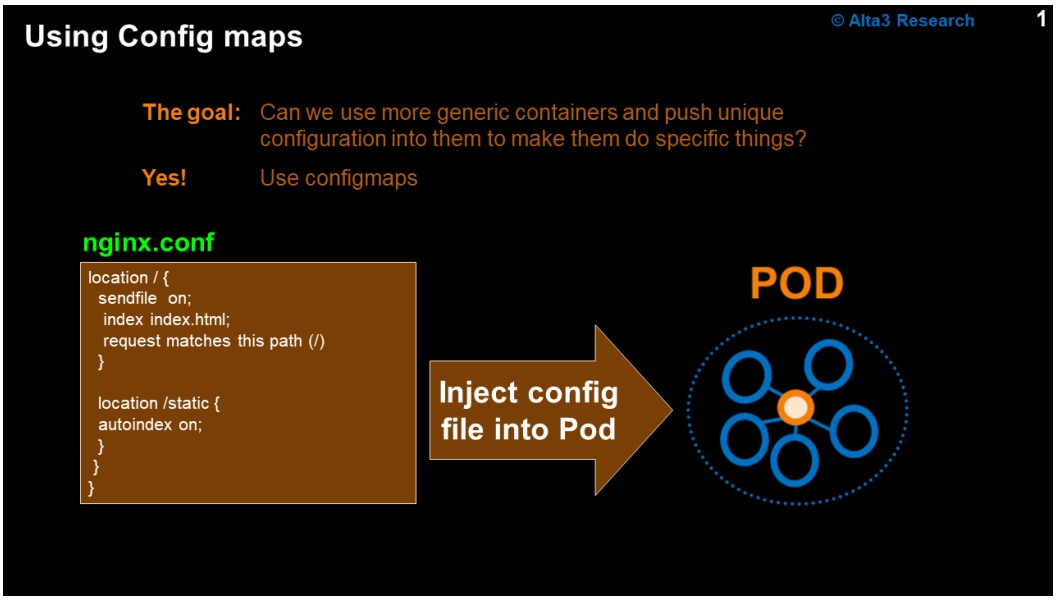
29. Advanced Logging

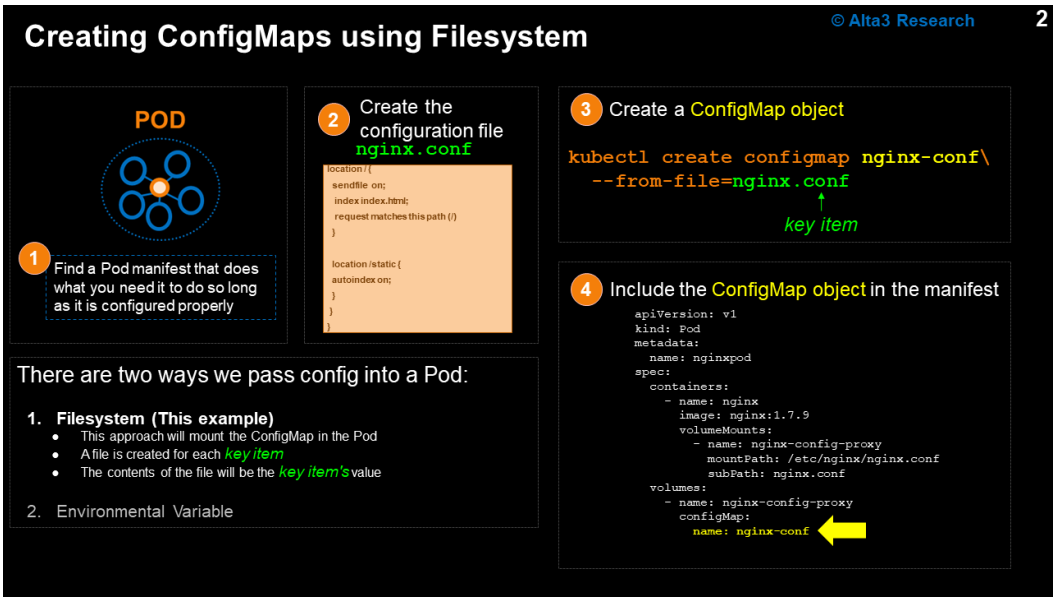






30. Configmaps





Adding a Volume to a Pod

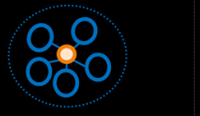
© Alta3 Research 3

```
apiVersion: v1
kind: Pod
metadata:
  name: nginxpod
spec:
  containers:
    - name: nginx
      image: nginx:1.7.9
      volumeMounts:
        - name: nginx-proxy-config
          mountPath: /etc/nginx/nginx.conf
          subPath: nginx.conf

  volumes:
    - name: nginx-proxy-config
      configMap:
        name: nginx-conf
```

- The name of the volume, now mounted in this container
- Mount location of the volume contents
- subPath is used ONLY when the specified mountPath directory exists; otherwise all files in that directory will be overwritten.
- The arbitrary name we gave the *volume* that now contains the configmap *ceph_ini*.
- The name of the configmap as it exists on our cluster

Creating ConfigMaps for Setting Environmental Variables

© Alta3 Research 4



1. Find a Pod manifest that does what you need it to do so long as it is configured properly

2. Create a ConfigMap object and define key/value pairs

```
kubectl create configmap ceph_env \
  --from-literal=moncount=3
  --from-literal=osdtype=ssd
```

There are two ways to pass config into a Pod:

1. Filesystem
2. **Environmental Variable (This example)**
   - This approach defines the env vars on the command line
   - An env var is created for each *key item*
   - The contents of the env var will be the *key item's* value

3. Include the ConfigMap object in the manifest

```
apiVersion: v1
. . .
spec:
  containers:
    - name: test-container
  . . .
      env:
        - name: MONCOUNT
          valueFrom:
            configMapKeyRef:
              name: ceph_env
              key: moncount
        - name: OSDTYPE
          valueFrom:
            configMapKeyRef:
              name: ceph_env
              key: osdtype
  restartPolicy: Never
```

Name of the env var

Name of the config map

31. Persistent Configuration with ConfigMaps

CKAD objective

- *Understand ConfigMaps*

Lab Objective

ConfigMaps allow Kubernetes Administrators to use more generic containers and pass specific configurations into them at the time of Pod creation.

In this lab, we will:

- Create three ConfigMaps.
- Learn how to manage ConfigMaps
- Assign the ConfigMapts to a Pod
- Develop a basic understanding of NGINX configuration, see https://hub.docker.com/_/nginx for more information

Outstanding Reference

Procedure

1. Setup your environment for this lab.

```
student@bchd:~$ setup persistent-configuration-with-configmaps
```

2. The first step with configmaps is to get organized. They get complicated very quickly, and versions spiral out of control. Let's do all our planning up front. We know we need to create three ConfigMaps. So we need a table to show the source and the object code.

Description	source	yaml	ConfigMap Name	Key
#1 nginx config file	~/mycode/config/nginx-base.conf	~/mycode/yaml/nginx-base-conf.yaml	nginx-base-conf	ngix.conf
#2 index.html	~/mycode/config/index-html-zork.html	~/mycode/yaml/index-html-zork.yaml	index-html-zork	index.html
#3 A static file	~/mycode/config/nineteen-eighty-four.txt	~/mycode/yaml/nineteen-eighty-four.yaml	nineteen-eighty-four	1984.txt

```
student@bchd:~$ batcat ~/mycode/config/nginx-base.conf
```

Click here to view the contents of **nginx-base.conf**

3. We need to convert this file to a ConfigMap object. Here is the command to do that.

```
student@bchd:~$ kubectl create configmap nginx-base-conf --from-file=nginx.conf=mycode/config/nginx-base.conf

configmap/nginx-base-conf created
```

4. The command you just completed above needs more attention. If you do not insert the KEY, then the file will assume the name of the source file, which would NEVER work. Many people have lost hours on this one! Always assign the key, which is simply the destination filename.

```
 --from-file=nginx.conf=mycode/config/nginx-base.conf
             ----------
                 ^
                 |
      Really? We need this KEY???
```

5. Now let's retrieve the yaml for that configmap with the following command.

```
student@bchd:~$ kubectl get configmaps nginx-base-conf -o yaml
```

Click here to view the contents of **nginx-base-conf.yaml**

6. Clearly a configmap is no longer something that we want to edit directly. So we will follow a convention just to stay organized!

    1. The source file will be stored in mycode/config/nginx-base-conf.conf
    2. The yaml file will be stored in mycode/yaml/nginx-base-conf.yaml
    3. When we edit the file, we start over with a new source and yaml pair.

7. We have already saved the above ConfigMap in ~/mycode/yaml/nginx-base-conf.yaml.

8. OK, let's move on to our second Configmap which is the index.html

Description	source	yaml	ConfigMap Name	Key
#1 nginx config file	~/mycode/config/nginx-base.conf	~mycode/yaml/nginx-base-conf.yaml	nginx-base-conf	ngix.conf
#2 index.html	~/mycode/config/index-html-zork.html	~mycode/yaml/index-html-zork.yaml	index-html-zork	index.html
#3 A static file	~/mycode/config/nineteen-eighty-four.txt	~mycode/yaml/nineteen-eighty-four.yaml	nineteen-eighty-four	1984.txt

9. Create the index.html ConfigMap. Look closely at the --from-file portion of the command below.

```
student@bchd:~$ kubectl create configmap index-html-zork  --from-file=index.html=mycode/config/index-html-zork.html

configmap/index-html-zork created
```

10. Now save the generated YAML file to index-html-zork.yaml.

```
student@bchd:~$ kubectl get configmaps index-html-zork -o yaml > index-html-zork.yaml
```

11. Let's do that one more time with the 1984 static text file.

```
student@bchd:~$ kubectl create configmap nineteen-eighty-four  --from-file=1984.txt=mycode/config/nineteen-eighty-four.txt
```

12. Now save the generated YAML file to nineteen-eighty-four.yaml.

```
student@bchd:~$ kubectl get configmap nineteen-eighty-four -o yaml > nineteen-eighty-four.yaml
```

13. List the ConfigMaps you have installed. You should see at least three.

```
student@bchd:~$ kubectl get configmap

NAME                 DATA   AGE
index-html-zork      1      19m
nginx-base-conf      1      3h37m
nineteen-eighty-four 1      134m
```

14. Study the `nginx-configured.yaml` Pod manifest. Note that our three ConfigMaps are included.

```
student@bchd:~$ batcat ~/mycode/yaml/nginx-configured.yaml
```

Click here to view the contents of **nginx-configured.yaml**

15. To what directory is index-html-zork being mapped?

    Answer
    /var/www/index.html

16. To what directory is nginx-base-conf being mapped?

    Answer
    /etc/nginx/nginx.conf

17. To what directory is nineteen-eighty-four being mapped?

    Answer
    /var/www/static/1984.txt

18. Start your NGINX pod.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/nginx-configured.yaml
```

19. Ensure your pod started.

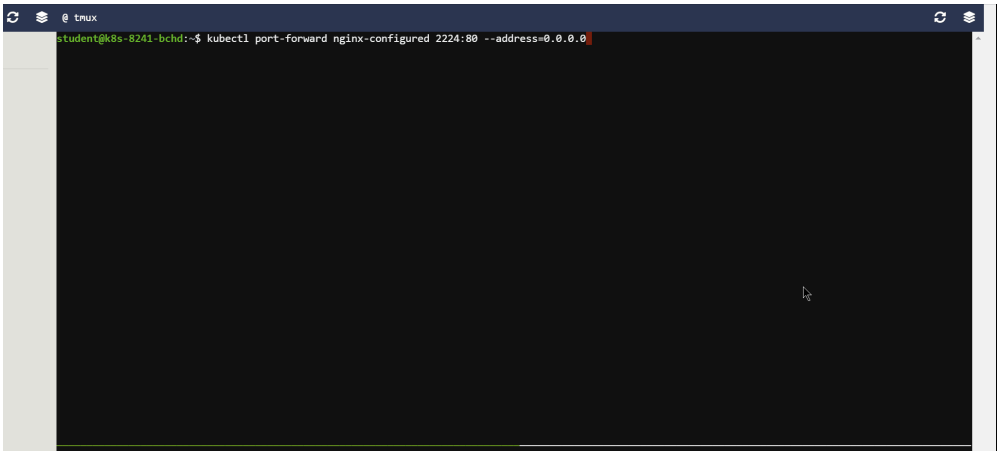
```
student@bchd:~$ kubectl get pod nginx-configured
```

20. Troubleshoot with the kubectl describe command.

21. Port forward the pod.

```
student@bchd:~$ kubectl port-forward nginx-configured 2224:80 --address=0.0.0.0
```

22. Click the right hand channel changer and select the `aux1` option, but to view your page, you need to go full screen, so click the pop-out icon. It is a blue box with an arrow.



23. You should be able to navigate to the file `1984.txt`, and see the text, "It was a bright cold day in April, and the clocks were striking thirteen."

24. When a directory and config file already exists, override like this using subpath:

```
containers:
- name: nginx
  volumeMounts:
    mountPath: /etc/nginx/nginx.conf
    subpath: nginx.conf
```

25. When a directory does NOT exist, plus you want to put a file in it, do this:

```
containers:
- name: nginx
  volumeMounts:
    mountPath: /var/www/static/nginx.txt
```

26. That's it for the main part of this lab, but check out the challenges below if your schedule permits.

Exam Review

https://kubernetes.io/docs/reference/generated/kubectl/kubectl-commands#-em-configmap-em-

https://kubernetes.io/docs/tasks/configure-pod-container/configure-pod-configmap/

1. **CHALLENGE 01 (OPTIONAL) -** Add a new ConfigMap that serves a txt file into `/var/www/static` that NGINX will serve.

- Create a txt file of your choosing. Save with a txt extension.
- Create a new ConfigMap.
- Edit the pod manifest to create a new volumeMount.
- Edit the pod manifest to create a matching volumes.
- Delete the old pod.
- Start this new pod.
- Port-forward the booted pod.
- Browse the new directory and file as a test.

1. *CHALLENGE 01 SOLUTION STEP* - Create a new txt file:

```
student@bchd:~$ vim new_file.txt
```

2. *CHALLENGE 01 SOLUTION STEP* - Place some text inside the file.

```
This is example
text. Please
add something
more interesting.
```

3. *CHALLENGE 01 SOLUTION STEP* - Save and exit with `:wq`

4. *CHALLENGE 01 SOLUTION STEP* - Create a new ConfigMap from that text file.

```
student@bchd:~$ kubectl create configmap new-file-cm --from-file=new_file.txt
```

5. *CHALLENGE 01 SOLUTION STEP* - Edit the pod manifest to look like the following.

```
---
apiVersion: v1
kind: Pod
metadata:
  name: nginx-configured
  labels:
    app: nginx-configured
spec:
  containers:
  - name: nginx
    image: nginx:1.18.0
    ports:
    - containerPort: 80
    volumeMounts:
    - name: nginx-proxy-config
      mountPath: /etc/nginx/nginx.conf
      subPath: nginx.conf
    - name: my-index-file
      mountPath: /var/www/index.html
      subPath: index.html
    - name: static-demo-data
      mountPath: /var/www/static/nginx.txt
      subPath: nginx.txt
    - name: challenge-config                    # this matches spec.volumes[3].name
      mountPath: /var/www/static/new_file.txt   # this is where the file appears INSIDE the pod
      subPath: new_file.txt                     # if this file already exists, overwrite it
  volumes:
  - name: nginx-proxy-config
    configMap:
      name: nginx-conf
  - name: my-index-file
    configMap:
      name: index-file
  - name: static-demo-data
    configMap:
      name: nginx-txt
  - name: challenge-config          # this is the name to be used in spec.containers.volumeMounts
    configMap:
      name: new-file-cm             # must match the configMap stored in etcd (created with kubectl)
```

6. *CHALLENGE 01 SOLUTION STEP* - Delete the old `nginx-configured` pod.

```
student@bchd:~$ kubectl delete pod nginx-configured
```

7. *CHALLENGE 01 SOLUTION STEP* - Create the new `nginx-configured` pod.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/nginx-configured.yaml
```

8. *CHALLENGE 01 SOLUTION STEP* - Port forward the newly configured nginx.

```
student@bchd:~$ kubectl port-forward nginx-configured 2224:80 --address=0.0.0.0
```

9. *CHALLENGE 01 SOLUTION STEP* - Browse the newly configured nginx pod using your channel changer to select the appropriate auxiliary port view that is now being served - similar to how you have previously been accessing your forwarded service.

32. Creating Ephemeral Storage For Fluentd Logging Sidecar

Fluentd is another Cloud Native Computing Foundation sponsored project which will allow you to collect data and send it out via a variety of output options.

In this lab, we will add a Fluentd container as a sidecar to a *logger* container that generates multiple logs. The *Fluentd container* will then stream the logs as one to whatever output we configure (*stdout* and via an *http* endpoint).

In order for our containers to share the files that are being logged, so that the *logger* can *produce* the logs, and the *Fluentd container* can *consume* the logs, we will house these logs inside of an **ephemeral volume** called an **emptyDir**.

emptyDir - An initially empty volume created when a Pod is assigned onto a node which allows all containers in the Pod to read and write to the files within it.

> Note that *emptyDir*, *configMap*, and *secrets* are some of the various types of ephemeral volumes that are available in Kubernetes.

Procedure

1. Run setup for this lab.

```
student@bchd:~$ setup fluentd
```

2. First thing that we will want to do is create the following ConfigMap. This creates the configuration file that Fluentd needs to know **what** files to have as a source, and **where** to send the logs. In this example, we are going to have *Fluentd* read the files `/var/log/1.log` & `/var/log/2.log` files from the *logger* container and sends these to **stdout**.

```
student@bchd:~$ batcat ~/mycode/yaml/fluentd-conf.yaml
```

Click here to view the contents of **fluentd-conf.yaml**

3. Now let's instantiate the ConfigMap.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/fluentd-conf.yaml
```

4. Next we will want to create the following Pod with two containers, a busybox container that generates logs, and a Fluentd container that aggregates these logs. Notice that they both are going to mount the single ephemeral volume **varlog**, which is an **emptyDir**. This creates a new directory that the Pods are both able to share. In this example, both the **count** and the **count-agent** container are mounting the **emptyDir** in their /var/log directories, but this would not be necessary. Inside each container the mount points are unique, even though they are both able to mount to the same attached volume.

```
student@bchd:~$ batcat ~/mycode/yaml/fluentd-pod.yaml
```

Click here to view the contents of **fluentd-pod.yaml**

5. Start your Pod!

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/fluentd-pod.yaml
```

6. Check the logs of the *count* container. We should not see any because all of the output is being streamed into the two files and *not* to stdout.

```
student@bchd:~$ kubectl logs logger -c count
```

> Nothing should be returned here

7. Now, check the logs of the *count-agent* container. This is the Fluentd image that is reaching across the volumes to stream the data from both of the files that the *count* container is writing to into its own *stdout* stream.

```
student@bchd:~$ kubectl logs logger -c count-agent
```

> Now you should see the logs from both of the files that the *count* container is writing into.

8. Now let's make this a little bit more interesting. We are going to update our configuration to point Fluentd to an external API that will "aggregate" the logs for us. To start with, let's update our ConfigMap to point to our BCHD machine.

```
student@bchd:~$ vim http_fluentd_config.yaml
```

Click here to view the contents of **http_fluentd_config.yaml**

9. Next we need to replace the `{{ BCHD_IP }}` field for our endpoint in our configuration file.

```
student@bchd:~$ sed -i "s/{{BCHD_IP}}/`nslookup bchd | grep Address | awk 'FNR==2{print $2}'`/g"  http_fluentd_config.yaml
```

10. Let's update our configmap now.

```
student@bchd:~$ kubectl apply -f http_fluentd_config.yaml
```

11. Now we are going to need to update the Fluentd image that we are using. The 1.30 version is far too outdated and does not have the *http* output plugin available by default. Let's use a newer image.

```
student@bchd:~$ vim http_fluentd.yaml
```

Click here to view the contents of **http_fluentd.yaml**

12. Create the following Python API to capture the HTTP logging that is being sent from our pod. This is going to be a "generic" example an HTTP based API logging software. There are many out there to choose from, so feel free to try to bring your own!

```
student@bchd:~$ vim logging-api.py

from flask import Flask, request

app = Flask(__name__)

@app.route("/logs", methods=["POST"])
def get_logs():
    log_data = request.data
    print(log_data)
    return "OKIE DOKIE"

if __name__ == "__main__":
    app.run(port=10001, host="0.0.0.0")
```

13. Before we can run our Python logger API, we will first need to install some Pythonic dependencies.

```
student@bchd:~$ python3 -m pip install flask
```

14. Start up your logging API.

```
student@bchd:~$ python3 logging-api.py
```

15. Now we are going to open up a **NEW TMUX PANE**. Prove that your Python code works by sending a message to your **bchd** logging API.

```
student@bchd:~$ curl -X POST -d '{"did-it-work": "you bet!"}' bchd:10001/logs -H "Content-Type: application/json"

OKIE DOKIE
```

16. Start your Pod.

```
student@bchd:~$ kubectl apply -f http_fluentd.yaml
```

17. Watch your python API get updated every 10 seconds!

```
    Huzzah! It works!
```

18. Once you have had your fun watching the *Fluentd* container send messages from your Pod to your *bchd* machine, it is time to clean up after ourselves. First, let's get rid of all of our pods.

```
student@bchd:~$ kubectl delete pods --all
```

19. Now, exit out of your **CURRENT TMUX PANE**.

```
student@bchd:~$ exit
```

20. Finish cleaning up by stopping your Python API logger with a **Ctrl C**.

You have just created two pods that each use a *Fluentd sidecar* to aggregate and stream your logs.

33. Secrets

K8s Secrets

© Alta3 Research 1

The goal: Can we move passwords, keys, security tokens and other data that is extra-sensitive into a Pod?

Yes! Use secrets.

Keys, Passwords, Security Tokens

Inject config file into Pod

POD

K8s Secrets

© Alta3 Research 2

1 Create a K8s secret

```
kubectl create secret generic web-tls \
  --from-file=alta3.key \
  --from-file=alta3.crt
```

3 Create the Pod

```
kubectl apply -f pod-with-secrets.yaml
```

4 This Pod now has access to the secrets in the /tls directory

2 Add the secret as a volume in the manifest.

pod-with-secrets.yaml

```
apiVersion: v1
kind: Pod
metadata:
  name: web-tls
spec:
  containers:
    - name: myweb-tls
      image: alta3.com/myweb-tls
      imagePullPolicy: Always
      volumeMounts:
      - name: tls-certs
        mountPath: "/tls"
        readOnly: true
  volumes:
    - name: tls-certs
      secret:
        secretName: web-tls
```

34. Create and Consume Secrets

Lab Objective

Secrets allow Kubernetes Administrators to encrypt and store sensitive data inside a Pod. With Kubernetes Secrets, you can store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys. Storing confidential information in a Secret is safer and more flexible than putting it verbatim in a Pod definition or in a container image.

In this lab, we will:

- Encrypt sensitive data and pass it into a Secret
- Mount the Secret as a volume in a Pod

Read more about Kubernetes Secretes here: https://kubernetes.io/docs/concepts/configuration/secret/

Secret Design Documentation: https://github.com/kubernetes/design-proposals-archive/blob/main/auth/secrets.md

Procedure

1. Run this script.

```
student@bchd:~$ setup create-and-consume-secrets
```

2. MySQL uses a login/password method for authentication. Since we need to keep the password secret, this is a perfect example of how to use the Kubernetes where Kind= Secret. Let's create the yaml to define our MySQL secret. The code speaks for itself. Note that the password is far too simple for the real world.

```
student@bchd:~$ batcat ~/mycode/yaml/mysql-secret.yaml
```

Click here to view the contents of **mysql-secret.yaml**

3. Now let's apply the secret.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/mysql-secret.yaml

secret/mysql-secret created
```

4. The next step is to write a pod manifest to that will use the above secret by injecting it into your pod as an environmental variable.

```
student@bchd:~$ batcat ~/mycode/yaml/mysql-locked.yaml
```

Click here to view the contents of **mysql-locked.yaml**

5. Now let's start the MySQL pod.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/mysql-locked.yaml
```

6. List the pods.

```
student@bchd:~/k8s-certs$ kubectl get pods
```

7. exec into the running pod.

```
student@bchd:~/k8s-certs$ kubectl exec --stdin --tty mysql-locked -- /bin/bash
```

8. Log into MySQL as follows:

```
root@mysql-locked:/# mysql -p
```

9. At the prompt, enter the following password:

```
alta3
```

Enter password:

Welcome to the MySQL monitor. Commands end with ; or \g. Your MySQL connection id is 1 Server version: 5.6.51 MySQL Community Server (GPL)

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>

10. Victory! But how do you exit? Type `exit;` to quit. The semicolon is required.

```
mysql> exit;

Bye
```

11. In the manifest, we set an environment variable for **MYSQL_ROOT_PASSWORD**, based on the secret we made at the beginning of the lab. Now that the password is on the pod, how secure is it? Let's find out.

```
root@mysql-locked:/# echo $MYSQL_ROOT_PASSWORD

alta3
```

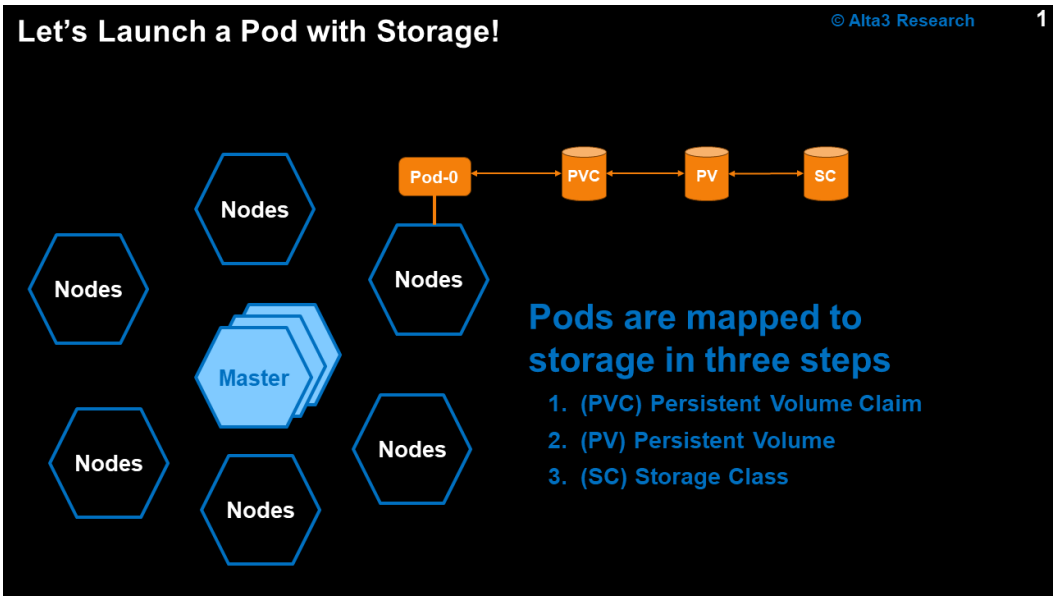
> It appears the password isn't very safe at all. This is the inherenet risk with using secrets in Kubernetes. Though there are methods to further safeguard our sensitive data using secrets, out-of-the-box; kubernetes secrets aren't very secret.

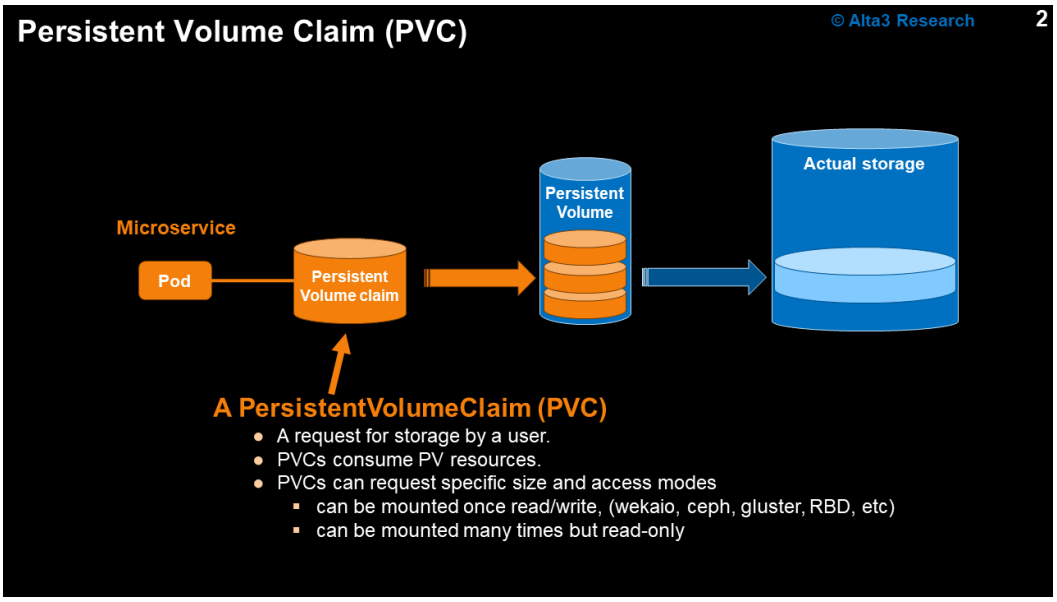
12. Exit the MySQL server.

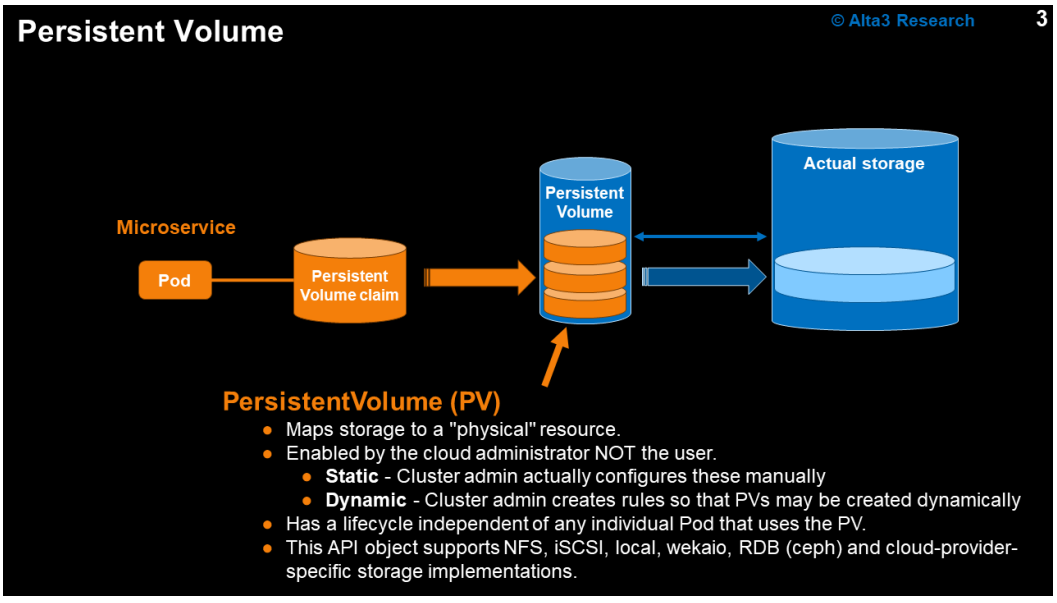
```
root@mysql-locked:/# exit
```

13. That's it for this lab. Good job! Be sure to `cd` back to the home directory for the next lab.

35. Storage Static



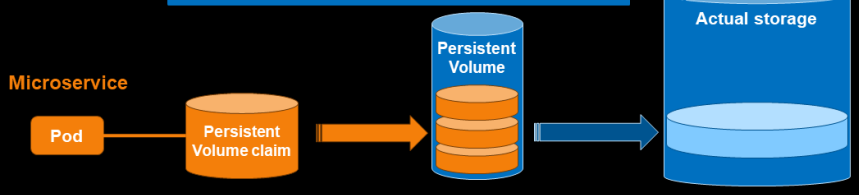




Storage Class

 4

Many types of physical storage are supported. The details of mapping a specific type of storage is handled in the PV (Persistent Volume) object yaml.

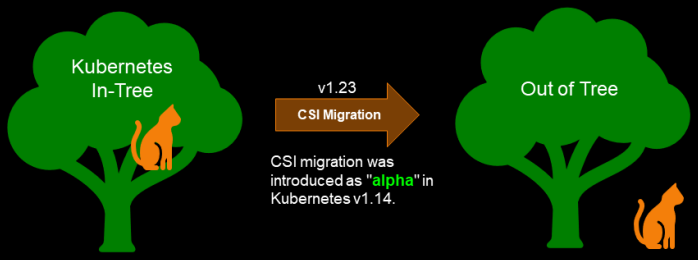


Storage Classes
Container Storage Interface

- AWSElasticBlockStore
- AzureFile
- AzureDisk
- CephFS
- Cinder
- FC
- FlexVolume
- Flocker
- GCEPersistentDisk
- Glusterfs
- iSCSI
- Quobyte
- NFS
- RBD
- VsphereVolume
- PortworxVolume
- ScaleIO
- StorageOS
- Local
- WEKAIO csi.weka.io

Container Storage Interface (CSI)

 5

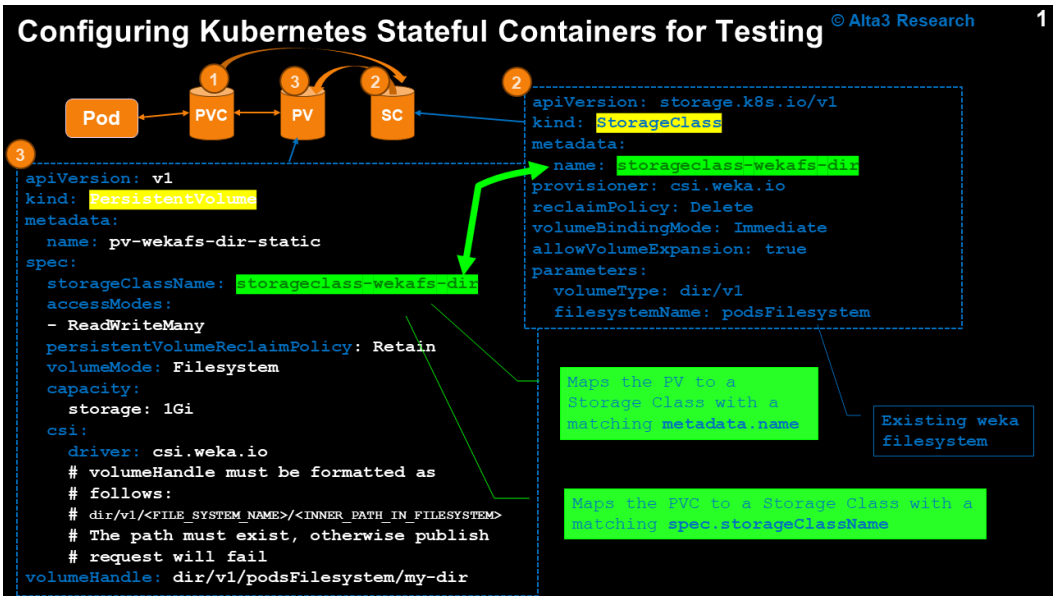


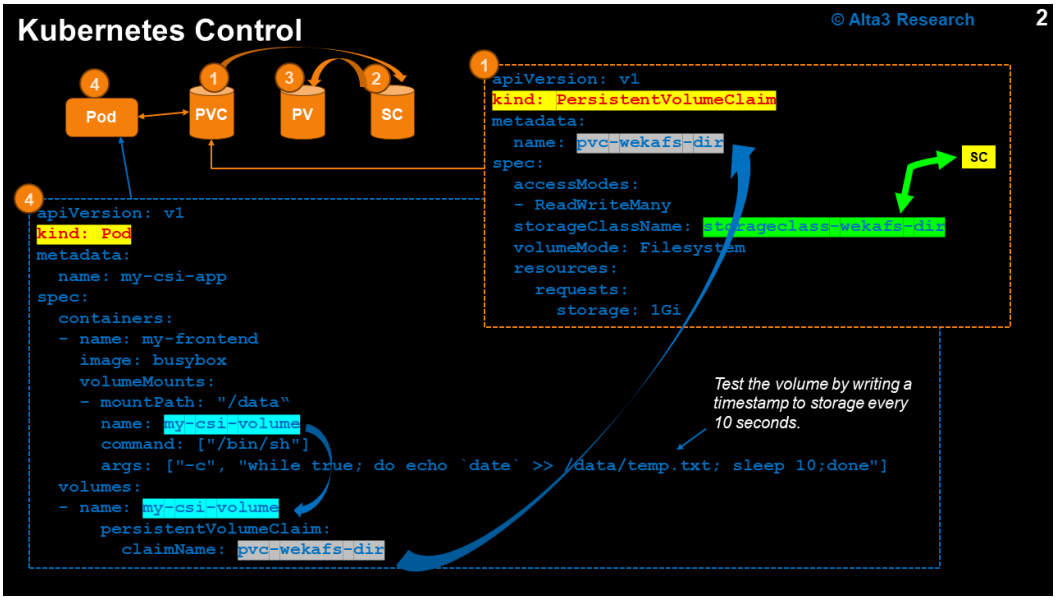
CSI migration was introduced as "alpha" in Kubernetes v1.14.

- Kubernetes core developers maintain storage service plugins in the Kubernetes code "tree"
- Too much work to keep current
- Vendor specific plugins a nightmare
- So, Kubernetes developers created the CSI.
- The CSI is an API, through which all storage vendors must attach.

- Kubernetes core developers will NOT maintain storage service plugins inside the Kubernetes code "tree", hence, out of tree.
- Vendors are required maintain storage service plugins
- Vendors must keep current to the CSI API.
- Want to use a specific vendor? Better check if they have written the driver!
- Want new features? You must use the CSI.

36. Storage Dynamic





37. Internal NFS via out-of-tree Container Storage Interface (CSI)

Objective:

- In this lab you will deploy an NFS server using the CSI and out of tree code. That means we are using the most modern way to deploy a storage access in Kubernetes. Because the NFS server is "internal" this means the NFS server will run internally on our cluster.
- Study an example of a complete CSI OOT k8s objects
- Configure an SC (storage class)
- Configure a PV
- Configure a PVC
- Write a sample file to the storage server we just created.
- Confirm that the file we just wrote exists on the storage server.
- Tear it all down

Procedure:

1. Deploy the NFS service first. Copy the entire code block below, drop it on the command line, and watch it run. We will do a forensic analysis after this runs. It will take a few seconds to install an NFS server as an in-cluster service plus the NFS CSI Drivers.

```
student@bchd:~$

export KUBE_NFS_CSI_REPO=https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/master/deploy
export KUBE_NFS_CSI_VERSION=v4.0.0
export KUBE_NFS_CSI_DRIVERS=${KUBE_NFS_CSI_REPO}/${KUBE_NFS_CSI_VERISON}/
kubectl apply -f ${KUBE_NFS_CSI_REPO}/example/nfs-provisioner/nfs-server.yaml
kubectl apply -f ${KUBE_NFS_CSI_DRIVERS}/rbac-csi-nfs.yaml
kubectl apply -f ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-driverinfo.yaml
kubectl apply -f ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-controller.yaml
kubectl apply -f ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-node.yaml
sleep 15
kubectl -n kube-system get pod -o wide -l app=csi-nfs-controller
kubectl -n kube-system get pod -o wide -l app=csi-nfs-node
kubectl get csidrivers
kubectl get services
```

2. Now let's check out what just happened. Look at the code block above and note that lines 4-8 install five objects. Just count the "kubectl apply -f" lines and you can see there are five of them. But what do they do? Let's move on and find out!

3. Start by using curl to check out the **FIRST** "kubectl apply" with the following command. REMEMBER, to escape batcat, type `q` when you have completed reviewing the file. The lab step following this one is analysis of this object.

```
student@bchd:~$ curl -L  ${KUBE_NFS_CSI_DRIVERS}/example/nfs-provisioner/nfs-server.yaml | batcat
```

```
  | ---
  | kind: Service  <-----
  | apiVersion: v1
  | metadata:
  |   name: nfs-server
  |   labels:
  |     app: nfs-server
  | spec:
  |   type: ClusterIP  # use "LoadBalancer" to get a public ip
  |   selector:
  |     app: nfs-server
  |   ports:
  |     - name: tcp-2049
  |       port: 2049
  |       protocol: TCP
  |     - name: udp-111
  |       port: 111
  |       protocol: UDP
  | ---
  | kind: Deployment  <----
  | apiVersion: apps/v1
  | metadata:
  |   name: nfs-server
... Truncated ...
```

4. Well now, that is interesting, this YAML is combining two object types that we already know! The first is a SERVICE, the second is a DEPLOYMENT which will call their creation "nfs-server." So you can conclude that the NFS service will run as a DEPLOYMENT, and the SERVICE will expose the deployment. Great, now that we found the actually NFS server, let's move on.

5. Use the following command to analyze the **SECOND** "kubectl apply". This installs FOUR objects. The results are shown below.

```
student@bchd:~$ curl -L  ${KUBE_NFS_CSI_DRIVERS}/rbac-csi-nfs.yaml | batcat
```

```
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: csi-nfs-controller-sa
  namespace: kube-system
---
apiVersion: v1
kind: ServiceAccount
metadata:
  name: csi-nfs-node-sa
  namespace: kube-system
---

kind: ClusterRole
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-external-provisioner-role
rules:
  - apiGroups: [""]
    resources: ["persistentvolumes"]
    verbs: ["get", "list", "watch", "create", "delete"]
  - apiGroups: [""]
    resources: ["persistentvolumeclaims"]
    verbs: ["get", "list", "watch", "update"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["storageclasses"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["events"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
  - apiGroups: ["storage.k8s.io"]
    resources: ["csinodes"]
    verbs: ["get", "list", "watch"]
  - apiGroups: [""]
    resources: ["nodes"]
    verbs: ["get", "list", "watch"]
  - apiGroups: ["coordination.k8s.io"]
    resources: ["leases"]
    verbs: ["get", "list", "watch", "create", "update", "patch"]
  - apiGroups: [""]
    resources: ["secrets"]
    verbs: ["get"]
---

kind: ClusterRoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: nfs-csi-provisioner-binding
subjects:
  - kind: ServiceAccount
    name: csi-nfs-controller-sa
    namespace: kube-system
roleRef:
  kind: ClusterRole
  name: nfs-external-provisioner-role
  apiGroup: rbac.authorization.k8s.io
```

6. Wow, this is a great example of an RBAC configuration. Note the creation of SERVICEACCOUNTs, CLUSTERROLE, and CLUSTERROLEBINDING. We already covered RBAC, so you should understand the mapping here. It is good to see that this project is managing access to the NFS server with RBAC. We would not want it any other way, would we? This may be handy for future reference, but for now, let's move on.

7. Use the following command to analyze the **THIRD** "kubectl apply".

```
student@bchd:~$ curl -L  ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-driverinfo.yaml | batcat
```

```
 1  | ---
 2  | apiVersion: storage.k8s.io/v1
 3  | kind: CSIDriver
 4  | metadata:
 5  |   name: nfs.csi.k8s.io
 6  | spec:
 7  |   attachRequired: false
 8  |   volumeLifecycleModes:
 9  |     - Persistent
10  |     - Ephemeral
11  |   fsGroupPolicy: File
```

8. The third "kubectl apply" installs the ***CSIDriver***! So that is what a CSI driver object looks like. Just 11 lines of code too! Let's move on.

9. Use the following command to analyze the **FOURTH** "kubectl apply".

```
student@bchd:~$ curl -L  ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-controller.yaml | batcat
```

```
 1  | ---
 2  | kind: Deployment
 3  | apiVersion: apps/v1
 4  | metadata:
 5  |   name: csi-nfs-controller
 6  |   namespace: kube-system
 7  | spec:
 8  |   replicas: 1
 9  |   selector:
10  |     matchLabels:
11  |       app: csi-nfs-controller
12  |   template:
13  |     metadata:
14  |       labels:
15  |         app: csi-nfs-controller
16  |     spec:
17  |       hostNetwork: true  # controller also needs to mount nfs to create dir
18  |       dnsPolicy: Default  # available values: Default, ClusterFirstWithHostNet, ClusterFirst
19  |       serviceAccountName: csi-nfs-controller-sa
20  |       nodeSelector:
21  |         kubernetes.io/os: linux  # add "kubernetes.io/role: master" to run controller on master node
TRUNCATED...
```

10. Where do we even start with this one? It's a deployment, but it looks different from other deployments we have seen. Fortunately, there are comments that tell use exactly what is going on here. On line 21, we see that this will run on the controller (master) node. That makes sense because this is a controller and we expect control software to run on the controller. Line 17's comment makes sense, as the controller is responsible to configure the NFS server. This reminds us that this object is going to manage the NFS server, not actually become an NFS server. We conclude that when we create a storage class, PVC, and PV this server will do the actual configuration of the NFS server to enable storage. Let's move on.

11. Use the following command to analyze the FIFTH (and final) "kubectl apply".

```
student@bchd:~$ curl -L  ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-node.yaml | batcat
```

```
1   | ---
2   | kind: DaemonSet
3   | apiVersion: apps/v1
4   | metadata:
5   |   name: csi-nfs-node
6   |   namespace: kube-system
TRUNCATED ....
```

12. So the FIFTH "kubectl apply" will install an NFS Controller DAEMONSET. That makes sense! We want the service available immediately on each node. A Daemonset installs this object on each node. We know what a Daemonset is, so let's move on.

13. Discover the internal cluster IP of the NFS service

```
student@bchd:~$ export KUBE_NFS_CLUSTER_IP=$(kubectl get services nfs-server -o template='{{.spec.clusterIP}}')
```

14. Confirm that you captured an IP address. This variable will be inserted into a PV object in just a few more steps.

```
student@bchd:~$ echo ${KUBE_NFS_CLUSTER_IP}

172.16.x.x
```

15. We are going to use a templating engine named Jinja2 to help fill out some manifests in the next few steps. This utility is GREAT. Just a few steps from now, all "j2 coolness" will be revealed.

```
student@bchd:~$ sudo apt install -y j2cli
```

16. Analyze the storage class YAML you are about to install. Note that the storage class references the new internal service.

```
student@bchd:~$ batcat ~/mycode/yaml/sc-nfs.yaml.j2
```

Click here to view the contents of **sc-nfs.yaml.j2**

17. Note the use of a jinja variable above: {{ KUBE_NFS_CLUSTER_IP }}. The double braces tell Jinja (j2) to insert the variable value here, replacing the double braces and the contents with the value of the variable ***KUBE_NFS_CLUSTER_IP***

18. Apply the storage class as follows, telling `j2` to handle the templating.

```
student@bchd:~$ kubectl apply -f <(j2 ~/mycode/yaml/sc-nfs.yaml.j2)
```

19. Confirm that the storage class is running:

```
student@bchd:~$ kubectl get sc

NAME      PROVISIONER      RECLAIMPOLICY   VOLUMEBINDINGMODE   ALLOWVOLUMEEXPANSION   AGE
nfs-csi   nfs.csi.k8s.io   Delete          Immediate           false                  13h
```

20. Let's analyze the PV manifest. We will be using Jinja to insert the IP address into the YAML.

```
student@bchd:~$ batcat ~/mycode/yaml/nginx-nfs-pv.yaml.j2
```

Click here to view the contents of **nginx-nfs-pv.yaml.j2**

21. Next, let's install the PV:

```
student@bchd:~$ kubectl apply -f <(j2 ~/mycode/yaml/nginx-nfs-pv.yaml.j2)
```

22. Let's look at a PVC manifest that will use our newly created PV.

```
student@bchd:~$ batcat ~/mycode/yaml/nginx-nfs-pvc.yaml
```

Click here to view the contents of **nginx-nfs-pvc.yaml**

23. Install the PVC we just analyzed:

```
student@bchd:~$ kubectl apply -f  ~/mycode/yaml/nginx-nfs-pvc.yaml
```

24. Now install a **POD** that uses the PVC we just created.

```
student@bchd:~$ batcat ~/mycode/yaml/nginx-nfs-example.yaml
```

Click here to view the contents of **nginx-nfs-example.yaml**

25. Instantiate the POD you just analyzed.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/nginx-nfs-example.yaml
```

26. Now exec into the POD named "nginx-nfs-example" to find the mount point.

```
student@bchd:~$ kubectl exec nginx-nfs-example -- bash -c "findmnt /var/www -o TARGET,SOURCE,FSTYPE"

TARGET   SOURCE        FSTYPE
/var/www 172.16.3.54:/ nfs4
```

27. OK, then ***/var/www*** it is! Let's write to that target.

```
student@bchd:~$ kubectl exec nginx-nfs-example -- bash -c "echo hello nurse! > /var/www/nfs-test"
```

28. The above steps stored ***"hello"*** (without the quotes) on the test client file called ***/var/www/nfs-test***, which will translate to ***/exports/nfs-test*** on the NFS server. In order to see if the file was written to the NFS server, you need to exec into the NFS pod and verify. Since the NFS server is part of a deployment, its pod has a dynamic pod name, you can use grep to discover the NFS storage pod and store the results in ***$NFS_POD_NAME***. Use the command to accomplish that task:

```
student@bchd:~$ export NFS_POD_NAME=`kubectl get pods | grep -o nfs-server[a-z0-9-]*`
```

29. Now that we have the pod name captured, exec into the NFS server and see if your file is there!

```
student@bchd:~$ kubectl exec $NFS_POD_NAME -- cat /exports/nfs-test
```

```
hello nurse!
```

30. Great! It works! That's it for this lab.

Tear down

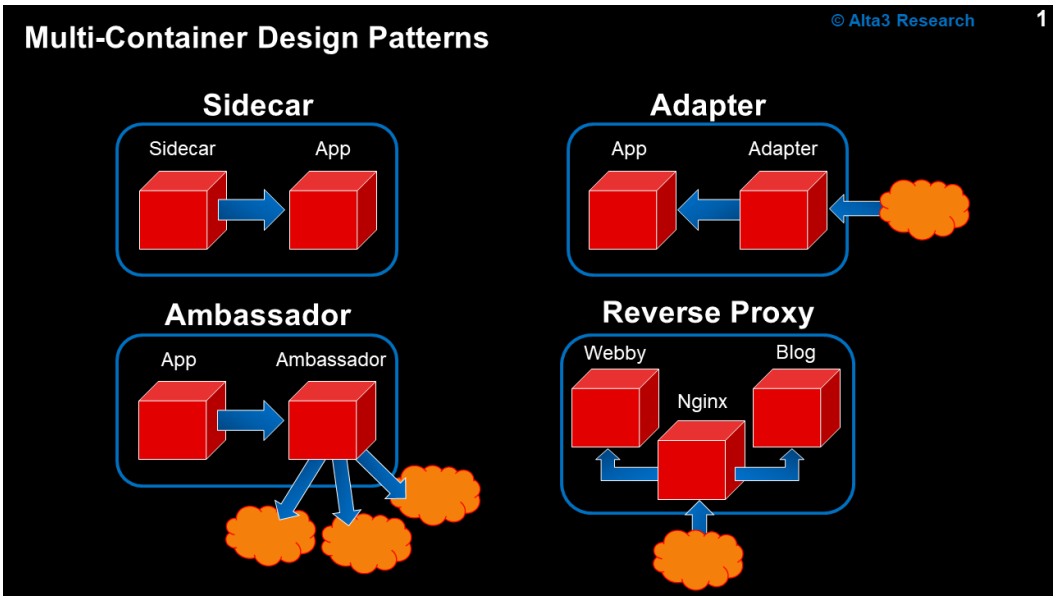
```
export KUBE_NFS_CSI_REPO=https://raw.githubusercontent.com/kubernetes-csi/csi-driver-nfs/master/deploy
export KUBE_NFS_CSI_VERSION=v4.0.0
export KUBE_NFS_CSI_DRIVERS=${KUBE_NFS_CSI_REPO}/${KUBE_NFS_CSI_VERISON}/
kubectl delete -f ${KUBE_NFS_CSI_REPO}/example/nfs-provisioner/nfs-server.yaml
kubectl delete -f ${KUBE_NFS_CSI_DRIVERS}/rbac-csi-nfs.yaml
kubectl delete -f ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-driverinfo.yaml
kubectl delete -f ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-controller.yaml
kubectl delete -f ${KUBE_NFS_CSI_DRIVERS}/csi-nfs-node.yaml
kubectl delete -f <(j2 mycode/yaml/nginx-nfs-pv.yaml.j2)
kubectl delete -f <(j2 mycode/yaml/sc-nfs.yaml.j2)
kubectl delete -f mycode/yaml/nginx-nfs-example.yaml
```

Note: in-cluster nfs fails name resolution

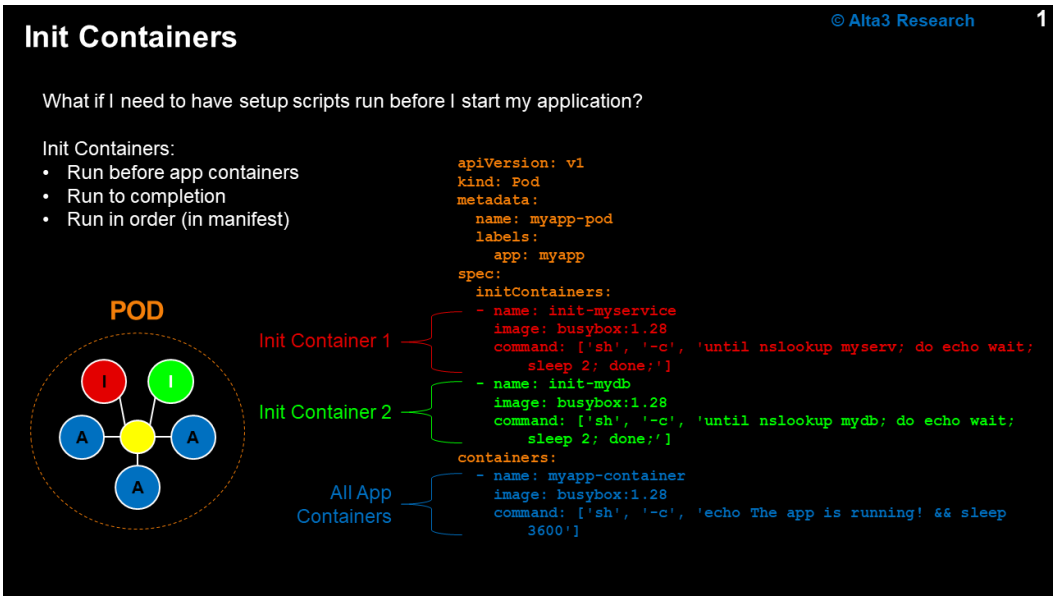
It is a known issue that the domain nfs-server.default.svc.cluster.local fails to resolve for deployments of nfs PV/PVC's (https://github.com/kubernetes/minikube/issues/3417)

- some workarounds: https://github.com/kubernetes/minikube/issues/3417#issuecomment-618068245
- our workaround: register and reference the server by ClusterIP

38. Sidecars



39. Init-Containers



40. Init Containers

Lab Objective

Create a Pod that uses Init Containers

Inside a Pod manifest, it is possible to include specialized **init containers** that will run before the rest of the app containers are started. These can be useful to run certain setup scripts that are not already present inside of an app image.

Unlike many regular app containers, init containers run to completion. Additionally, there can be more than one init container specified inside of a Pod. If so, then the first init container listed will run to completion, then the second will run to completion, and so on. This will continue until all of the init containers have successfully completed, *then* the app containers will start to run.

Procedure

1. Run setup for this lab

```
student@bchd:~$ setup init-containers
```

2. Let's create a Pod that has two init containers inside of it. The first init container, `init-myservice`, wants to resolve `myservice` via `nslookup`. The second init container, `init-mydb`, wants to resolve `mydb`. Our container, `myapp-container` will not be able to launch until these two init containers are able to successfully run to completion.

```
student@bchd:~$ batcat ~/mycode/yaml/init-cont-pod.yaml
```

Click here to view the contents of **init-cont-pod.yaml**

3. Instantiate the Pod.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/init-cont-pod.yaml

pod/myapp-pod created
```

4. Verify that the Pod has been scheduled. Also, take note of the `STATUS` of the Pod. **Init:0/2** is stating that there are still two init containers that have not yet run to completion. This is expected, as neither init container can successfully complete their tasks (resolving `myservice` and `mydb`).

```
student@bchd:~$ kubectl get pods

NAME                                READY   STATUS     RESTARTS   AGE
myapp-pod                           0/1     Init:0/2   0          5s
```

5. Because we have the first init container waiting to discover a service via the **nslookup** command, it will continue to check for it every two seconds. Once found, it will complete. So let's check the logs of our first init container.

```
student@bchd:~$ kubectl logs myapp-pod -c init-myservice

...
waiting for myservice
nslookup: can't resolve 'myservice'
nslookup: can't resolve 'myservice'
Server:    172.16.3.10
Address 1: 172.16.3.10 kube-dns.kube-system.svc.cluster.local

waiting for myservice
nslookup: can't resolve 'myservice'
Server:    172.16.3.10
Address 1: 172.16.3.10 kube-dns.kube-system.svc.cluster.local
```

6. Now that we see it has been attempting to find **myservice**, let's give it that service to discover.

```
student@bchd:~$ vim myservice.yaml
```

7. Create the following Kubernetes resource, called a "Service". A Service in Kubernetes is an abstraction which defines a logical set of Pods and a policy by which to access them. Services enable a loose coupling between dependent Pods. A Service is defined using YAML (preferred) or JSON, like all Kubernetes objects. The set of Pods targeted by a Service is usually determined by a LabelSelector (see below for why you might want a Service without including selector in the spec). Although each Pod has a unique IP address, those IPs are not exposed outside the cluster without a Service. Services allow your applications to receive traffic. In short, when we create this service, it will associate an IP address with our init-myservice Pod. It should look like the below:

Click here to view the contents of **myservice.yaml**

8. Initiate **myservice**.

```
student@bchd:~$ kubectl apply -f myservice.yaml

service/myservice created
```

9. For now, we're not focused on exploring services, however you can see some information about the new service resource by typing the following:

```
student@bchd:~$ kubectl get services
```

10. See how `myservice` has an IP address associated with it? Now, let's look at the status of the Pod.

```
student@bchd:~$ kubectl get pods

NAME                                READY   STATUS     RESTARTS   AGE
myapp-pod                           0/1     Init:1/2   0          5m
```

11. View the logs from the first init container now. Notice how the final line of the log shows that it has found the **myservice** address.

```
student@bchd:~$ kubectl logs myapp-pod -c init-myservice

...
nslookup: can't resolve 'myservice'
Server:    172.16.3.10
Address 1: 172.16.3.10 kube-dns.kube-system.svc.cluster.local

Name:      myservice
Address 1: 172.16.3.47 myservice.default.svc.cluster.local
```

12. Now that our Pod has successfully had one init container run to completion, let's have the other one do so as well. Create another service manifest, this time for the **mydb** service.

```
student@bchd:~$ vim mydb.yaml
```

Click here to view the contents of **mydb.yaml**

13. Start up the **mydb** service.

```
student@bchd:~$ kubectl apply -f mydb.yaml

service/mydb created
```

14. Now, let's check the logs of the second init-container.

```
student@bchd:~$ kubectl logs myapp-pod -c init-mydb

...
waiting for mydb
Server:    172.16.3.10
Address 1: 172.16.3.10 kube-dns.kube-system.svc.cluster.local

Name:      mydb
Address 1: 172.16.3.135 mydb.default.svc.cluster.local
```

15. Excellent! Our two init containers have successfully performed their tasks. We should now see that our regular app container is in a **Running** state.

```
student@bchd:~$ kubectl get pods

NAME                                READY   STATUS     RESTARTS   AGE
myapp-pod                           1/1     Running    0          7m
```

Good work! We just created an app that waited to verify two services were running prior to starting up its primary containers using two unique **init containers**!

41. Labels

The Purpose of Labels

 1

Labels

- "Tags" for objects
- Used for grouping, viewing, and operating many objects at the same time
- The primary means that k8s objects are identified and collected!

Applying Labels

 2

```
apiVersion: v1
kind: Pod
metadata:
  name: testpod
  labels:
    runtime: webby
    ver: 2
    ssd: true
    env: dev
spec:
  containers:
  - name: nginx
    image: nginx:1.18
    ports:
    - containerPort: 80
```

Modifying Labels

 3

Adding or Modifying a Label

```
kubectl label deployments nginx "canary=true"
```

Removing a Label

```
kubectl label deployments nginx "canary-"
```

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

API Object Label Selectors

 4

matchLabels: (AND) Selecting all Pods with the label *app=flamingo & slap=alpaca*

```
selector:
  matchLabels:
    app: flamingo
    ver: 2
```

matchExpressions: (OR) Select all Pods where the key is app and the value is *in* the following list: alpaca or flamingo

```
selector:
  matchExpressions:
    - { key: app, operator: In, values: [alpaca,
flamingo]}
```

Label Selectors

 5

the key **ver** has the value **2**

```
kubectl get pods --selector="ver=2"
```

the key **app** has the value of **flamingo**
&&
the key **ver** has the value of **2**

logical "and"

```
kubectl get pods --selector="app=flamingo,ver=2"
```

the key **app** has either the value of **alpaca || flamingo**

logical "or"

```
kubectl get pods --selector="app in (alpaca, flamingo)"
```

42. Understanding Labels and Selectors

CKAD Objective

- *Understand how to use Labels, Selectors, and Annotations*

Lab Objective

In this lab, we will be attaching labels to Pods and Deployments, and then using these labels to select Pods and Deployments.

Organization with Kubernetes can mean the difference between "Success!" and "Oh, no..."

Labels enable users to map their own organizational structures onto system objects in a loosely coupled fashion, without requiring clients to store these mappings. Objects, remember, are those things we are creating within Kubernetes. The pod "webby", for example, is a Pod object.

Because labels are searchable, they offer an obvious advantage when trying to identify similar builds (versions), apps, or purposes. In short, labeling allows for an easy search and view of objects you or your organization would like to link together.

Labels are just like social media hash-tags, but they include a "key:value" pair (displayed in the kubectl output as "key=value"). Both the "key" and the "value" are dynamic (you can set them however you wish). However, **do not ever expose secret information with labels**, such as passwords (i.e. `pass=qwerty123`) or even usernames. Stick to things like, "silo=5g-research" or "app=ng-controller".

Additionally, labels allow Kubernetes resources to be **selected** by other Kubernetes resources. For example, a ReplicaSet chooses which Pod(s) to maintain based upon the labels that the Pod(s) have placed on them.

Questions

- **What labels would you use in your company?**
  - *This one is mostly a thought experiment, but answers might include things like, 'app=firewall', 'app=acme-sbc' or 'app=homebrewed-automation', 'silo=5g', 'silo=customersales', 'enddate=10-2021'*
- **What makes labeling important when working in a Kubernetes environment?**
  - *Usability will be reduced as complexity increases if organization is not preserved. Labels aid in organizing the cluster.*

Procedure

1. Run setup for this lab.

```
student@bchd:~$ setup understanding-labels-and-selectors
```

2. Let's begin by returning to the home directory.

```
student@bchd:~$ cd
```

3. Let's take a quick look at what contexts are available to our kubectl client. Remember, a context is a reference to a cluster, namespace, and user.

```
student@bchd:~$ kubectl config get-contexts
```

4. Set kubectl so it controls the namespace `default`, within our cluster, as the user `admin`. We gave this configuration the name, `kubernetes-the-alta3-way` context.

```
student@bchd:~$ kubectl config use-context kubernetes-the-alta3-way
```

5. Delete any extra deployments that may still be running.

```
student@bchd:~$ kubectl get deployments
```

```
student@bchd:~$ kubectl delete deployment <INSERT_DEPLOYMENT_NAME_HERE>
```

> Remember, a deployment is a higher-level concept than a "pod". A deployment is asking Kubernetes to manage pods, exposing features like "rolling updates", "rollbacks", and "replicas". In production, there is no doubt you'll use deployments. However, in testing (or when you're first learning), a more basic level on off pod is sufficient.

6. Delete any extra pods and deployments that may still be running.

```
student@bchd:~$ kubectl get pods
```

```
student@bchd:~$ kubectl delete pod <INSERT_POD_NAME_HERE>
```

7. The following command will delete all pods and deployments in the current namespace. It is a shortcut for the commands you just ran.

```
student@bchd:~$ kubectl delete pod,deploy --all
```

8. Start the pod `nginx` using a manifest available from our GitHub repository.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/nginx-pod.yaml
```

9. Get the labels on every pod. Notice that NGINX has the label `what=what` currently.

```
student@bchd:~$ kubectl get pods --show-labels --all-namespaces

NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE    LABELS
default       nginx                                      1/1     Running   0          110s   what=what
kube-system   calico-kube-controllers-dd597647d-k74cl    1/1     Running   0          42h    k8s-app=calico-kube-controllers,pod-template-
hash=dd597647d
kube-system   calico-node-fvbsx                          1/1     Running   0          42h    controller-revision-hash=6794b8f589,k8s-app=calico-
node,pod-template-generation=1
kube-system   calico-node-jgxch                          1/1     Running   0          42h    controller-revision-hash=6794b8f589,k8s-app=calico-
node,pod-template-generation=1
kube-system   calico-node-sz2gm                          1/1     Running   0          42h    controller-revision-hash=6794b8f589,k8s-app=calico-
node,pod-template-generation=1
kube-system   coredns-68567cdb47-mqhhd                   1/1     Running   0          42h    k8s-app=kube-dns,pod-template-hash=68567cdb47
kube-system   coredns-68567cdb47-whgqg                   1/1     Running   0          42h    k8s-app=kube-dns,pod-template-hash=68567cdb47
```

10. Get the version label of all pods with label `app`. The value of `app` is our key, which will be displayed as a new column. If there are any matches on this label, the value will appear in the column.

```
student@bchd:~$ kubectl get pods -L app

NAME      READY     STATUS    RESTARTS   AGE       APP
nginx     1/1       Running   1          1m
```

11. Looks like `nginx` does not have a key for the value `app`. Let's apply a new label to `nginx`. We'll assign the key `app` the value of `web_service`.

```
student@bchd:~$ kubectl label pods nginx app=web_service
```

12. Get the label value of all pods with label key `app`.

```
student@bchd:~$ kubectl get pods -L app

NAME      READY     STATUS    RESTARTS   AGE       APP
nginx     1/1       Running   1          3m        web_service
```

13. Now we should also be able to update the manifest to accomplish this. Perform the following `get` command to get the running pod's manifest.

```
student@bchd:~$ kubectl get pod nginx -o yaml > nginx-pod-update.yaml
```

14. Now edit the YAML with your favorite text editor (vi, vim, nano, etc.)

```
student@bchd:~$ vim nginx-pod-update.yaml
```

15. Make the following edits. **Remove** the label `what=what` and make your label section match what's listed below.

```
...
metadata:
  annotations:
    kubectl.kubernetes.io/last-applied-configuration: ...
  creationTimestamp: "2022-05-12T15:17:26Z"
  labels:
    app: web_service
    importance: critical
    customer: internal_use
  name: nginx
...
```

16. Save and exit with `:wq`

17. Now apply the configuration change to your pod.

```
student@bchd:~$ kubectl apply -f nginx-pod-update.yaml
```

```
pod/nginx configured
```

18. Once again, get the labels on every pod.

```
student@bchd:~$ kubectl get pods --show-labels --all-namespaces

NAMESPACE     NAME                                       READY   STATUS    RESTARTS   AGE     LABELS
default       nginx                                      1/1     Running   1          5m27s
app=web_service,customer=internal_use,importance=critical
kube-system   calico-kube-controllers-7798c85854-g24wg   1/1     Running   0          3h38m   k8s-app=calico-kube-controllers,pod-template-
hash=7798c85854
kube-system   calico-node-m4x6v                          1/1     Running   0          3h38m   controller-revision-hash=6b8fb7c5fd,k8s-app=calico-
node,pod-template-generation=1
kube-system   calico-node-s2k4n                          1/1     Running   0          3h38m   controller-revision-hash=6b8fb7c5fd,k8s-app=calico-
node,pod-template-generation=1
kube-system   calico-node-s9q22                          1/1     Running   0          3h38m   controller-revision-hash=6b8fb7c5fd,k8s-app=calico-
node,pod-template-generation=1
kube-system   coredns-5d65dd49c8-9fvrz                   1/1     Running   0          3h38m   k8s-app=kube-dns,pod-template-hash=5d65dd49c8
kube-system   coredns-5d65dd49c8-jh2bp                   1/1     Running   0          3h38m   k8s-app=kube-dns,pod-template-hash=5d65dd49c8
```

19. Did you notice that `nginx` now has three labels (key:value pairs)?

```
app=web_service,customer=internal_use,importance=critical
```

The first one we applied using `kubectl`, and the second two were added by editing the manifest.

20. Get the label value of all pods with label key `app` (yes we did this already, just confirm our old label was not overwritten).

```
student@bchd:~$ kubectl get pods -L app

NAME      READY     STATUS    RESTARTS   AGE       APP
nginx     1/1       Running   0          6m        web_service
```

21. Get the label value of all pods with label key `customer`.

```
student@bchd:~$ kubectl get pods -L customer

NAME      READY     STATUS    RESTARTS   AGE       CUSTOMER
nginx     1/1       Running   0          7m        internal_use
```

22. Get the label value of all pods with label key `importance`.

```
student@bchd:~$ kubectl get pods -L importance

NAME      READY     STATUS    RESTARTS   AGE       IMPORTANCE
nginx     1/1       Running   0          8m        critical
```

23. Delete the pod to clear our cluster. Let's use the local manifest we updated to do so.

```
student@bchd:~$ kubectl delete -f nginx-pod-update.yaml
```

Label a Kubernetes Pod

> Kubernetes objects are integral parts of the system, but telling one from the other can be difficult. That is what labels and annotations are for! Labels are key/value pairs that are attached to objects- their purpose is to give meaning and relevance to the user. They also allow users to organize objects for greater efficiency. Labels can be added when objects are created but can be modified whenever you wish. These labels must have a unique key/value label in order to work.

24. Create an NGINX deployment. Create the following `Deployment` manifest. Deployments are higher level than "Pods". A Deployment is controlled by Kubernetes, and allows for cool features like, image upgrades and rollbacks.

```
student@bchd:~$ vim ~/nginx-obj.yaml
```

Click here to view the contents of **nginx-obj.yaml**

25. Save and exit with `:wq`

26. Let's compare and contrast a Pod to a Deployment. Create both a single stand alone Pod, and a deployment, within your cluster (and current namespace).

```
student@bchd:~$ kubectl apply -f ~/nginx-obj.yaml
```

27. Now create a stand alone Pod from the manifest available via the HTTP URL.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/nginx-pod.yaml
```

28. List all of the pods. A single stand alone pod will "only" have its name. A pod that is part of a deployment is also given a UUID trailer to uniquely identify the Pods as belonging to a replicated set. This replication is handled by Kubernetes' Replication Controller.

```
student@bchd:~$ kubectl get pods

NAME                                READY     STATUS              RESTARTS   AGE
nginx                               1/1       Running             0          9h
nginx-obj-create-6c54bd5869-8bqxt   1/1       Running             0          10m
nginx-obj-create-6c54bd5869-npj82   1/1       Running             0          10m
nginx-obj-create-6c54bd5869-tk424   1/1       Running             0          10m
```

29. Add a label to a pod. This is a simple `label` command tacked onto the kubectl. For `<FULLPODNAME>` choose one of the three pods in the nginx-obj manifest and put the full name of the pod. Example: `nginx-obj-create-6c54bd5869-tk424`.

```
student@bchd:~$ kubectl label pods <FULLPODNAME> project=easy
```

30. Show the label.

```
student@bchd:~$ kubectl get pods -L project
```

31. Add another label to a different pod (again by full name).

```
student@bchd:~$ kubectl label pods <FULLPODNAME> segment=alpha
```

32. Show both labels.

```
student@bchd:~$ kubectl get pods -L project,segment

NAME                                READY     STATUS    RESTARTS   AGE       PROJECT   SEGMENT
nginx                               1/1       Running   0          9m
nginx-obj-create-6c54bd5869-grhpd   1/1       Running   0          9m        easy
nginx-obj-create-6c54bd5869-phq9f   1/1       Running   0          9m                  alpha
nginx-obj-create-6c54bd5869-zrs2w   1/1       Running   0          9m
```

33. Clean out the cluster.

```
student@bchd:~$ kubectl delete -f ~/nginx-obj.yaml

student@bchd:~$ kubectl delete -f https://static.alta3.com/projects/k8s/nginx-pod.yaml

student@bchd:~$ kubectl get pods # should be empty
```

Remove Pod Labels

> Labels are VERY important when it comes to organizing your Kubernetes environment. When it comes to labels, searching for an issue could be the difference of *minutes* (if you have labels) and *hours* (if you don't have labels)! Nobody likes slow response times!

34. Use the YAML doc `nginx-pod.yaml` to create a pod.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/nginx-pod.yaml
```

35. Ensure that the pod was created and retrieve a list of pod names.

```
student@bchd:~$ kubectl get pods
```

36. Now apply the tag `project=wrong`.

```
student@bchd:~$ kubectl label pods nginx project=wrong
```

37. Review the values for the label `project` as applied to the current pods.

```
student@bchd:~$ kubectl get pods -L project
```

38. Oh no! You added an incorrect label to a pod. Let's remove it now! FYI, the confirmation that the label has been removed isn't terribly helpful. It might be more accurate if their message simply indicated that a label change has occurred.

```
student@bchd:~$ kubectl label pods nginx "project-"
```

```
pod/nginx unlabeled
```

39. Confirm that the label has been removed.

```
student@bchd:~$ kubectl get pods -L project

NAME                              READY     STATUS              RESTARTS   AGE       PROJECT
nginx                             1/1       Running             0          3m
```

40. Try creating your own label and then removing it using the methodology outlined in this lab.

41. Use the YAML doc `nginx-pod.yaml` to remove the pod.

```
student@bchd:~$ kubectl delete -f ~/mycode/yaml/nginx-pod.yaml
```

Apply, Modify, and Search for Labels

> Kubernetes was made to grow with you as your application scales both in size and complexity. With this in mind, labels and annotations were added as a foundational concept. Labels are key/value pairs that can be attached to Kubernetes objects such as Deploys, Pods and ReplicaSets. They can be arbitrary and are useful for attaching identifying information to Kubernetes Objects.

42. Create the alpaca-prod Pod. Run the following:

```
student@bchd:~$ kubectl run alpaca-prod --image=gcr.io/kuar-demo/kuard-amd64:1 --labels="ver=1,app=alpaca,env=prod"

pod "alpaca-prod" created
```

43. Create the alpaca-test Pod. Run the following:

```
student@bchd:~$ kubectl run alpaca-test --image=gcr.io/kuar-demo/kuard-amd64:2 --labels="ver=2,app=alpaca,env=test"

pod "alpaca-test" created
```

44. Create bandicoot-prod Pod. Run the following:

```
student@bchd:~$ kubectl run bandicoot-prod --image=gcr.io/kuar-demo/kuard-amd64:2 --labels="ver=2,app=bandicoot,env=prod"

pod "bandicoot-prod" created
```

45. Create the bandicoot-staging Pod. Run the following:

```
student@bchd:~$ kubectl run bandicoot-staging --image=gcr.io/kuar-demo/kuard-amd64:2 --labels="ver=2,app=bandicoot,env=staging"

pod "bandicoot-staging" created
```

46. Now list your pods, showing the labels.

```
student@bchd:~$ kubectl get pods --show-labels

NAME                READY   UP-TO-DATE   AVAILABLE   AGE   LABELS
alpaca-prod         1/1     1            1           45s   app=alpaca,env=prod,ver=1
alpaca-test         1/1     1            1           32s   app=alpaca,env=test,ver=2
bandicoot-prod      1/1     1            1           22s   app=bandicoot,env=prod,ver=2
bandicoot-staging   1/1     1            1           9s    app=bandicoot,env=staging,ver=2
```

47. We've worked with labels before, but this additional practice is demonstrating how they might be applied within a corporate environment. Just substitute `alpaca` and `bandicoot` for your in-house mission critical app(s). Labels can also be applied or updated on objects after they are created. If you update a label you have on a deployment, it will only update the deployment in this lab, not the object itself.

48. Update the label you made from the previous steps. In our test environment, we're trialing some alerting code called `canary`. Set a tag so we know this test environment is running the code.

```
student@bchd:~$ kubectl label pods alpaca-test "canary=true"

pods "alpaca-test" labeled
```

49. Now check out the change here. Labels make it quite a bit easier to track what is going on within our environment.

```
student@bchd:~$ kubectl get pods --show-labels
```

50. Let's tag that our production machines are both firewalled. We can label both of these in one step.

```
student@bchd:~$ kubectl label pods alpaca-prod bandicoot-prod "firewall=true"
```

51. Looks like it worked! Ensure our changes were felt.

```
student@bchd:~$ kubectl get pods --show-labels

NAME                DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE       LABELS
alpaca-prod         2         2         2            2           29m       app=alpaca,env=prod,firewall=true,ver=1
alpaca-test         1         1         1            1           27m       app=alpaca,canary=true,env=test,ver=2
bandicoot-prod      2         2         2            2           20m       app=bandicoot,env=prod,firewall=true,ver=2
bandicoot-staging   1         1         1            1           20m       app=bandicoot,env=staging,ver=2
```

52. Label selectors are used to filter Kubernetes objects based on a set of labels. They are used both by end users (via tools like kubectl) and by different types of objects (such as how ReplicaSet relates to its pods). We showed all the labels of all our deployments in the last lab. Let's list deploys with a certain version now.

53. Check out the `--selector` option associated with `kubectl get pods` (it can also be used with other `kubectl` commands). You'll find the following:

```
student@bchd:~$ kubectl help get pods | less

-l, --selector='': Selector (label query) to filter on, supports '=', '==', and '!='.(e.g. -l key1=value1,key2=value2)
```

54. Run the *--show-labels* again to see the labels we have.

```
student@bchd:~$ kubectl get pods --show-labels
```

55. Now use the `selector` to get the version or label we want.

```
student@bchd:~$ kubectl get pods --selector=ver=2

NAME                READY   STATUS    RESTARTS
AGE                                                                                |·
alpaca-test         1/1     Running   0
61s                                                                                |·
bandicoot-prod      1/1     Running   0
46s                                                                                |·
bandicoot-staging   1/1     Running   0         39s
```

56. Now find pods that are labeled as being within staging.

```
student@bchd:~$ kubectl get pods --selector=env=staging

NAME                READY   STATUS    RESTARTS
AGE                                                                                |·
bandicoot-staging   1/1     Running   0         110s
```

57. We can use the bang-equals `!=` to indicate exclusion. Let's find all pods not labeled staging.

```
student@bchd:~$ kubectl get pods --selector=env!=staging

NAME             READY   STATUS    RESTARTS
AGE                                                                                  |·
alpaca-prod      1/1     Running   0
3m10s                                                                                |·
alpaca-test      1/1     Running   0
2m46s                                                                                |·
bandicoot-prod   1/1     Running   0
2m31s                                                                                |·
nginx            1/1     Running   0         7m
```

58. Let's clean things up. Again, we can start employing shortcuts by combining commands. In this command, we'll remove **four** deployments at once.

```
student@bchd:~$ kubectl delete pods alpaca-prod alpaca-test bandicoot-staging bandicoot-prod

pod "alpaca-prod" deleted
pod "alpaca-test" deleted
pod "bandicoot-prod" deleted
pod "bandicoot-staging" deleted
```

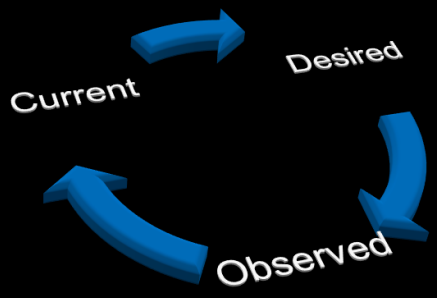
59. Great! That's it for this lab.

43. Replicasets

Reconciliation Loops

 1

A process to keep the current, desired, and observed states the same



ReplicaSets Features

 2

ReplicaSets are a separate and autonomous API from Pods

- ReplicaSets use the same process you do to create Pods…
- …they just do it automatically!

Adopt existing Pods

- Have a running Pod that you want to assign to a ReplicaSet to keep it running?
- Just create a ReplicaSet for that Pod!

Scalable Microservices and ReplicaSets

 3

The Good

- Adding or deleting a Pod makes no functional difference
- Ideal for stateless services
- Typically ReplicaSets are fronted by a load balancer

The Bad

- Works poorly for stateful services
- So keep your persistent data outside of ReplicaSets.

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

ReplicaSet Spec

ReplicaSets

- Defined with a specification
- Must have a unique name
- Defines number of replicas
- A Pod template (so that ReplicaSet controller can rebuild a missing Pod)

```
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: test-replicaset
spec:
  replicas: 3
  selector:
    matchLabels:
      tier: frontend
  template:
    metadata:
      labels:
        tier: frontend
    spec:
      containers:
      - name: nginx
        image: nginx
```

ReplicaSet Commands

Delete Pods created by ReplicaSets and also the ReplicaSet:

```
kubectl delete rs nginx
```

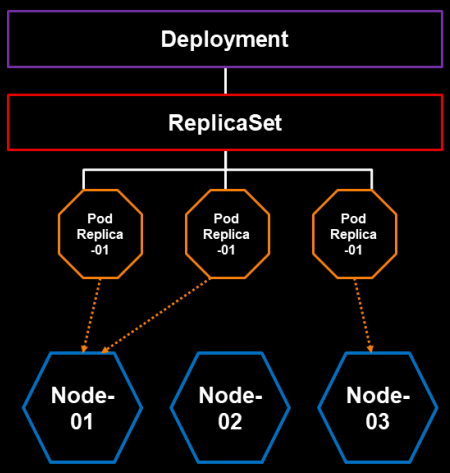
Delete ReplicaSets object but KEEP THE PODS:

```
kubectl delete rs nginx --cascade=orphan
```

44. Deployments Manifests

Deployments

© Alta3 Research 1



A Deployment

- Pods are RARELY launched on a cluster by themselves
- Pods are OFTEN launched as a deployment
- Deployments may declare how many replicas of a Pod should be running at a time.

Advantages over ReplicaSets:

- Does everything a ReplicaSet can do, PLUS:
  - Version control (roll out, roll back)
  - Zero down time
  - Accepts any changes made to yaml, EXPECTS change!

Deployment Internals

© Alta3 Research 2

Manages new releases

- **Manages rollouts** (version upgrades)
- **Can fallback** to previous version (version downgrade)
- New versions are incrementally deployed, verified, put into production until the entire ReplicaSet is upgraded.

45. Writing a Deployment Manifest

CKAD Objective

- *Understand Deployments and how to perform rolling updates*

Lab Objective

- Effortlessly delete Pods and Deployments with or without a manifest.
- Create a Deployment for webby.

Deployments are higher-level than ReplicaSets. Think of a deployment as a wrapper around a ReplicaSet. Therefore, just like a ReplicaSet, a deployment will ensure that there is a "correct" number of Pods available (that the deployed count matches `replica:`).

Deployments also allow for tracking revision history while performing rollouts or rollbacks. This is a feature that ReplicaSets by themselves lack.

For more reading on Kubernetes Deployments, check out: https://kubernetes.io/docs/concepts/workloads/controllers/deployment/

Questions

Name at least two ways to launch a pod.

- *Use the command, "kubectl run"*
- *Create a manifest file and the command, "kubectl create -f " or "kubectl apply -f "*
  - *Create a kind: Pod*
  - *Create a kind: Deployment*
- *Looking ahead, we'll learn that using Helm is also a viable technique.*

What are the differences in the ways to launch a pod or Kubernetes object?

- *Using a command at the CLI is fast, but work is not recorded*
- *Using a manifest is a way to easily record work with a version control software, like git and GitHub*
- *kind: Pod - launches a single pod*
- *kind: Deployment - launches 1 or more pods that may be replicated and updated with new images via a 'rollout'*
- *Using automation, like Helm, allows for rapid, predictable, continuous deployment of complete systems*

Additional Reading:

- kubectl apply vs kubectl create: https://stackoverflow.com/questions/47369351/kubectl-apply-vs-kubectl-create
- [Imperative vs Declarative] Imperative vs Declarative: https://medium.com/bitnami-perspectives/imperative-declarative-and-a-few-kubectl-tricks-9d6deabdde

Procedure

1. Let's view a manifest for us that is already written. You'll see it creates a Deployment for 'zombie' with a replica count of 3.

```
student@bchd:~$ batcat ~/mycode/yaml/zombie.yaml
```

Click here to view the contents of **zombie.yaml**

2. Launch Alta3's zombie.yaml to see how a Deployment behaves in your cluster:

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/zombie.yaml

deployment "zombie" created
```

3. You see that there are three zombie pods running. They may have different names than the entry below.

```
student@bchd:~$ kubectl get pods

NAME                      READY     STATUS    RESTARTS   AGE
zombie-6c54bd5869-8nn6f   1/1       Running   0          23s
zombie-6c54bd5869-9m8z4   1/1       Running   0          22s
zombie-6c54bd5869-zh77f   1/1       Running   0          23s
```

4. Delete the first one as follows (your index value will be different).

```
student@bchd:~$ kubectl delete pod zombie-6c54bd5869-8nn6f
```

5. INSTANTLY, you run a `get` command:

```
student@bchd:~$ kubectl get pods

NAME                      READY     STATUS              RESTARTS   AGE
zombie-6c54bd5869-9m8z4   0/1       Terminating         0          1h
zombie-6c54bd5869-fxcq8   1/1       Running             0          29s
zombie-6c54bd5869-tfpnr   0/1       ContainerCreating   0          2s
zombie-6c54bd5869-zh77f   1/1       Running             0          1h
```

6. **DELETE USING THE SOURCE YAML** "The easiest way to delete an object is to reference the same YAML that made it in the first place. So this should work...":

```
student@bchd:~$ kubectl delete -f ~/mycode/yaml/zombie.yaml

deployment.apps "zombie" deleted
```

7. You relaunch the deployment, `zombie.yaml`

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/zombie.yaml

deployment.apps/zombie created
```

8. Another way to delete a Deployment is to `get` a listing of the Deployments, then delete the Deployment by its name:

```
student@bchd:~$ kubectl get deployment

NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
zombie    3         3         3            3           4m
```

9. As it turns out, you may also **delete using a deployment name**. To delete the entire deployment, use the deployment keyword, like this:

```
student@bchd:~$ kubectl delete deployment zombie

deployment "zombie" deleted
```

10. If you check, you will notice that the zombie pods are all gone. This may take a few seconds for all of the pods to terminate.

```
student@bchd:~$ kubectl get pods

No resources found.
```

11. Create your zombie deployment again.

```
student@bchd:~$ kubectl create -f ~/mycode/yaml/zombie.yaml
```

12. Show the labels of the zombie deployment.

```
student@bchd:~$ kubectl get deployment zombie --show-labels

NAME      DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE       LABELS
zombie    3         3         3            3           50s       app=nginx
```

13. **DELETE WITH LABEL-** Delete the entire deployment like this:

```
student@bchd:~$ kubectl delete deploy -l app=nginx

deployment.apps "zombie" deleted
```

14. Copy this deployment framework into your favorite editor.

```
student@bchd:~$ vim webby-deploy.yaml
```

Click here to view the contents of **webby-deploy.yaml**

15. Using the above framework, convert your webby pod into a deployment that creates three replicas. The name of the deployment, any labels, and the port number can be decided by you. Look below at the solution if you need help or want to double check your work.

    CHALLENGE

16. Start your webby deployment.

    CHALLENGE

17. Delete your webby-deployment using YAML and CONFIRM the deployment is deleted.

CHALLENGE

18. Start your webby-deployment up again. Then, delete your webby-deployment using the deployment name and CONFIRM the deployment is deleted.

    CHALLENGE

19. Start your webby-deployment up again. Then, delete your webby-deployment using the label you assigned and CONFIRM the deployment is deleted.

    CHALLENGE

46. The Answers

```
---
apiVersion: apps/v1
kind: Deployment
metadata:
   name: webservice
   labels:
     app: webby
spec:
  selector:
    matchLabels:
      app: webby
  replicas: 3
  template:
    metadata:
      labels:
        app: webby
    spec:
      containers:
      - name: webby
        image: registry.gitlab.com/alta3/webby:latest
        ports:
        - containerPort: 8888

student@bchd:~$ kubectl apply -f webby-deploy.yaml

deployment.apps/webservice created

student@bchd:~$ kubectl delete -f webby-deploy.yaml

deployment "webservice" deleted

student@bchd:~$ kubectl apply -f webby-deploy.yaml

deployment.apps/webservice created

student@bchd:~$ kubectl delete deployment webservice

deployment "webservice" deleted

student@bchd:~$ kubectl apply -f webby-deploy.yaml

deployment.apps/webservice created

student@bchd:~$ kubectl delete deployment -l app=webby

deployment "webservice" deleted
```

47. Deployments Rollout

Recreate vs RollingUpdate

 1

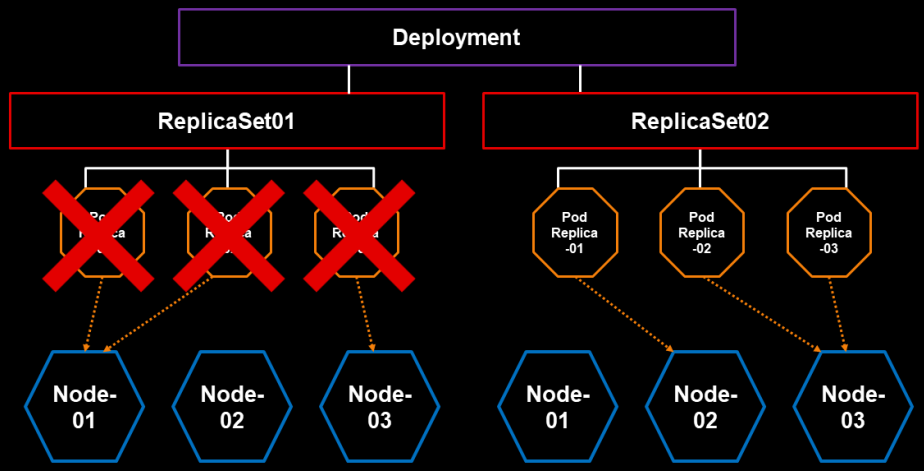
Kubernetes Rollout strategies

1. Recreate
   - Simple
   - Really fast
   - Good for testing
   - Creates downtime so bad idea for production
   - A failed rollout is catastrophic

2. RollingUpdate
   - Slower than recreate
   - Much more robust
   - No down time
   - Failed rollout has a small blast radius

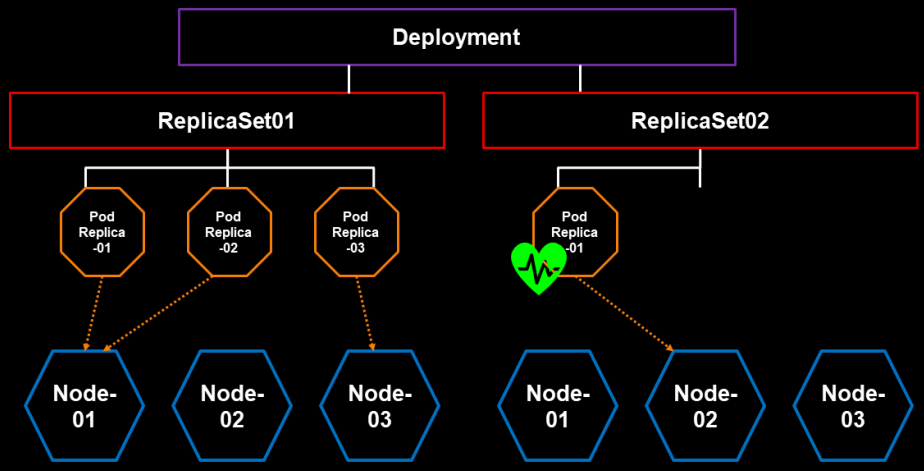
Deployment Rollout Strategy: RECREATE

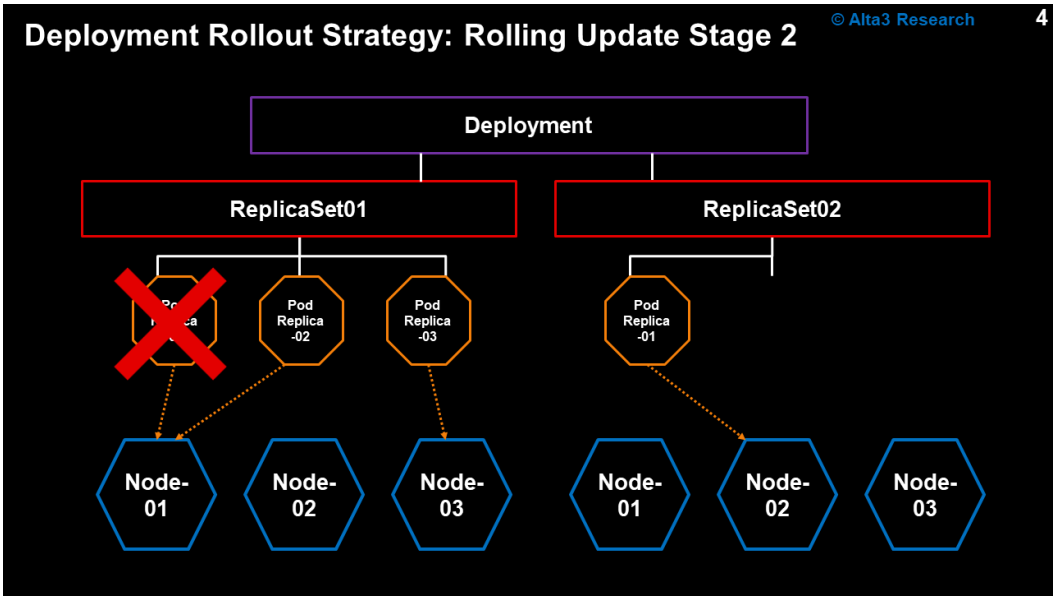
 2

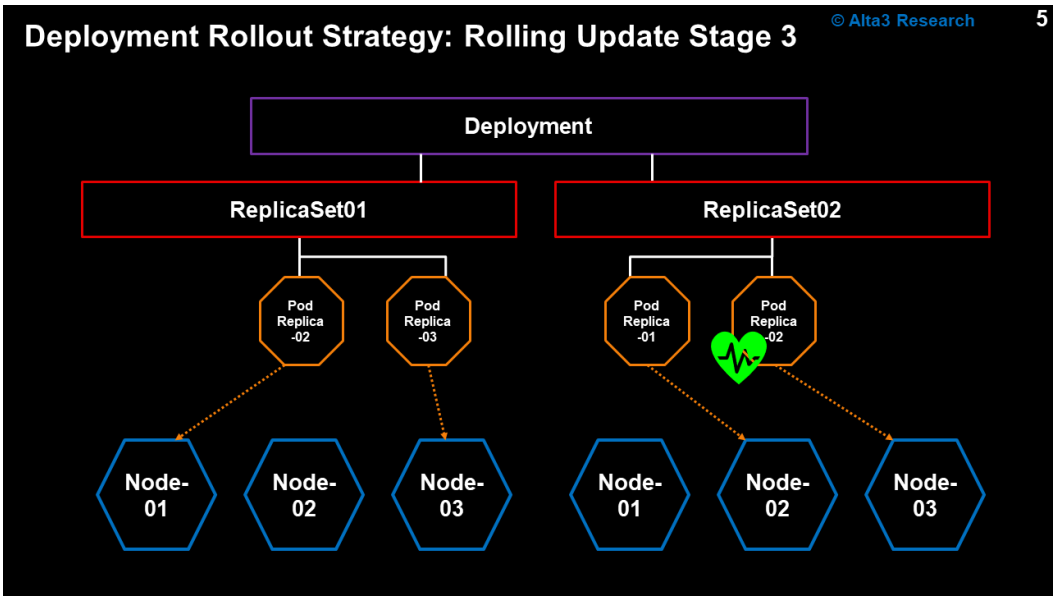


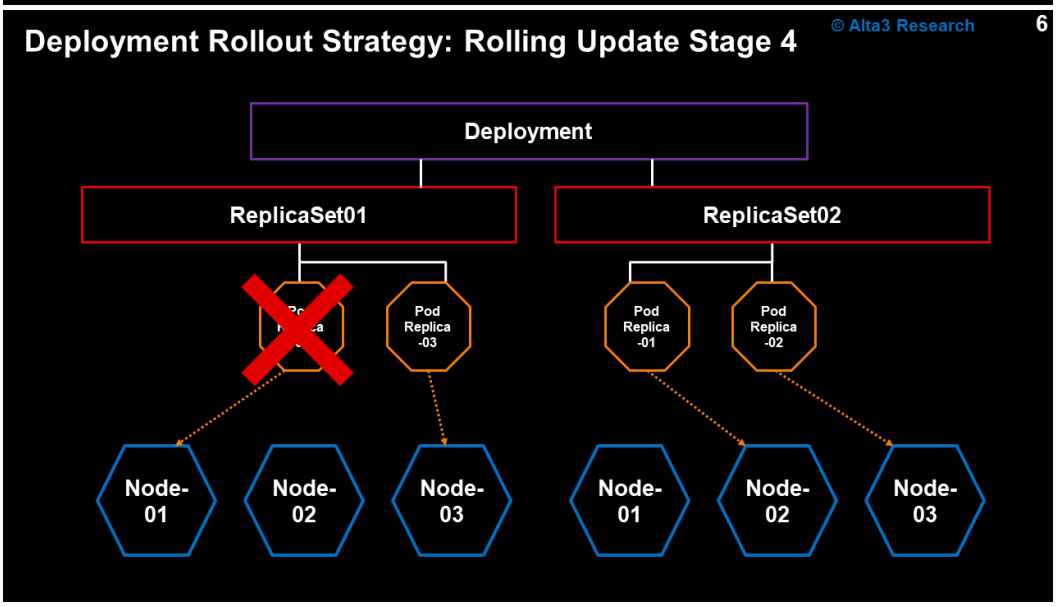
Deployment Rollout Strategy: Rolling Update Stage 1

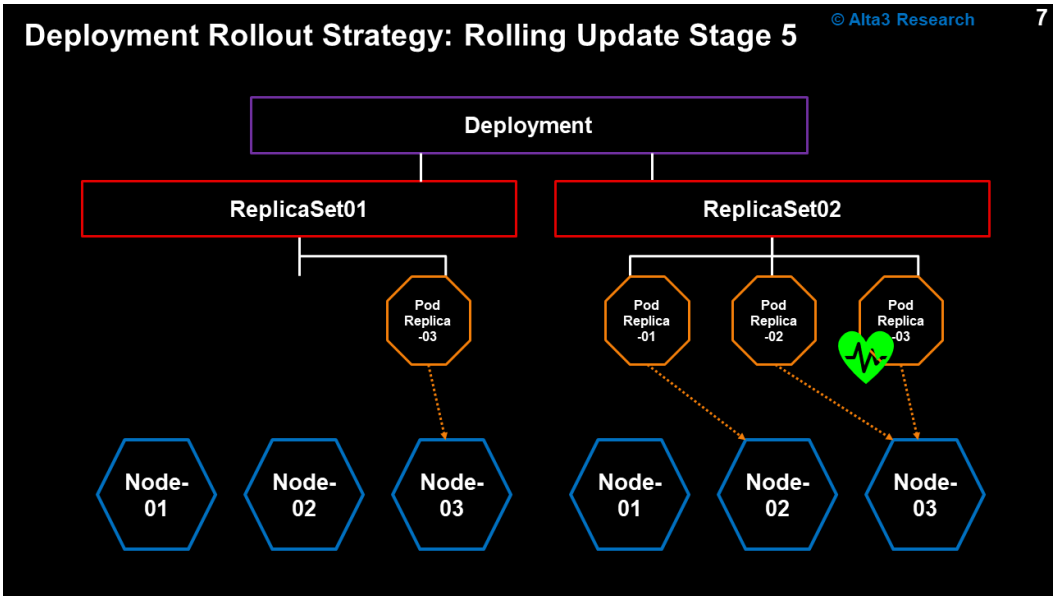
 3

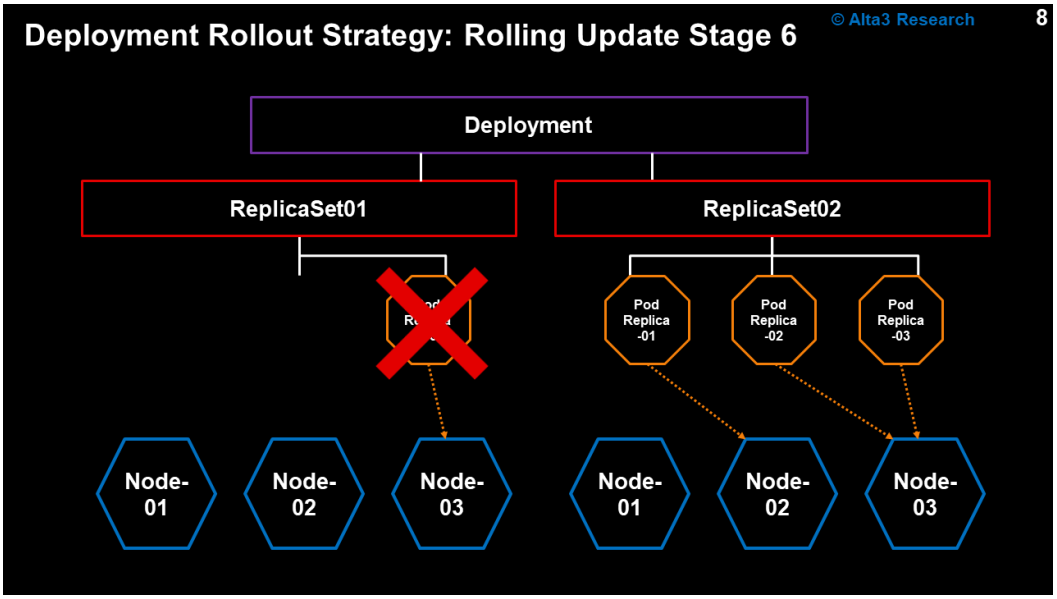


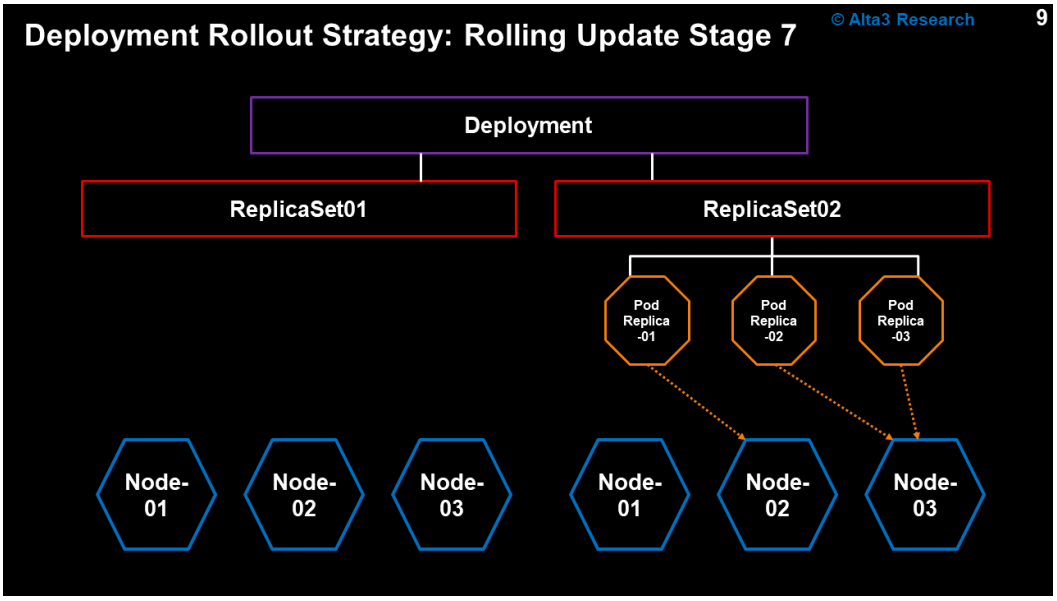












Updating a deployment

 10

```
kubectl apply -f nginx-dpymnt.yaml
    ...
    spec:
      strategy:
        rollingUpdate:
          maxSurge: 1
          maxUnavailable: 1
        type: RollingUpdate
      template:
        metadata:
          name: nginx-deploy
        spec:
          containers:
          - image: nginx:2.0.1
            imagePullPolicy: Always
    ...
```

If the manifest has changed, then using *kubectl apply* will trigger a rollout

Staged Rollout

 11

- ✓ The idea is a simple, just slow things down.
- ✓ Make sure the new service is healthy first, then rollout the next

```
...
  spec:
    minReadySeconds: 60
...
```

- ✓ Kill a hanging rollout with timeout

```
...
  spec:
    progressDeadlineSeconds: 600
...
```

kubectl rollout

 12

Display the rollout history like this:

```
kubectl rollout history deployment <name>
```

Display the rollout history details for specific version:

```
kubectl rollout history deployment <name> --revision=3
```

UNDO! - At any point in the rollout.

```
kubectl rollout undo deployment <name>
```

Return to a specific version:

```
kubectl rollout undo deployment <name> --to-revision=2
```

Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com

48. Rolling Updates and Rollbacks

CKAD Objectives

- *Understand Deployments and how to perform rolling updates*
- *Understand Deployments and how to perform rollbacks*

Lab Objective

- The objective of this lab is to learn to rollout and rollback deployments while observing best-practice skills (documenting your work with annotations). Within this lab you'll create a deployment of the 'zombie' image with a replica count of 3. Update the running image within the 'zombie' via a 'rollout.' Observe each of the pods within the deployment update in real time. Finally, rollback the deployment to the previous version.

Read more about Kubernetes updates here:
https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#in-place-updates-of-resources

This lab uses annotation to keep rollouts organized. Read more about annotations here: https://kubernetes.io/docs/concepts/overview/working-with-objects/annotations/

Procedure

ROLLOUTS

1. To start, we want to split our screen in the horizontal direction. Press `CTRL+b`, take your hands off the keyboard, then press `SHIFT+'` (i.e. create a double-quote).

2. In the new screen, run the following command.

```
student@bchd:~$ watch -n 2 'kubectl get pods'
```

3. You now want to switch panes. Press `CTRL+b` take your hands off the keyboard, then press the directional UP arrow to move to the pane "above" your current pane.

4. In the "top" pane, create a zombie.yaml file so you know what you are deploying.

```
student@bchd:~$ vim zombie.yaml
```

Click here to view the contents of **zombie.yaml**

5. Looks like `zombie.yaml` creates a deployment. If you're not clear on what a deployment is, you can check out the official documentation at https://kubernetes.io/docs/concepts/workloads/controllers/deployment/ Once you're up to date, run the following command.

```
student@bchd:~$ kubectl apply -f zombie.yaml

deployment.apps/zombie created
```

6. Verify your deployment is alive. In the bottom pane, you will see the new pods being created.

```
student@bchd:~$ kubectl get deployments

NAME     READY   UP-TO-DATE   AVAILABLE   AGE
zombie   3/3     3            3           31s
```

7. Take a look at the pods.

```
student@bchd:~$ kubectl get pods
```

8. Consider the following question:

   - **Why are there three zombie-yyyyyyyyyy-zzzzz pods?**
     - *Because a Deployment is made up of 1 or more pods. In this case replicas=3, so three identical pods are being deployed*
   - **Is there any reason zombie-yyyyyyyyyy-zzzzz is named the way it is?**
     - *Yes! All pods within a deployment are given this "double-hyphen" notation. Anytime you observe two-hyphens within a pod name, you can assume it is part of a deployment.*

9. Take a look at the rollout history. You should only see one.

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         <none>
```

10. Set an annotation on the **CHANGE-CAUSE** field.

```
student@bchd:~$ kubectl annotate deployment zombie kubernetes.io/change-cause="deployed ver. 1.18.0"

deployment.apps/zombie annotated
```

11. Once again look at the rollout history. There is still only one, but now we have a tiny comment letting us know why our pod was deployed.

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         deployed ver. 1.18.0
```

12. Get the names of the pods and choose one of these to use for the next step.

```
student@bchd:~$ kubectl get pods

NAME                      READY   STATUS    RESTARTS   AGE
zombie-5bf87f5f59-bhnx4   1/1     Running   0          9m27s
zombie-5bf87f5f59-bpfxk   1/1     Running   0          9m27s
zombie-5bf87f5f59-zr6ft   1/1     Running   0          9m27s
```

13. Take a look inside one of your zombie pods to check the deployed version.

```
student@bchd:~$ kubectl exec -it zombie-<replace-with-zombie-pod-id> -- bash
```

14. Once you are inside your container, check the version of NGINX that is running.

```
root@zombie-<pod-suffix>:/# nginx -v

nginx version: nginx/1.18.0
```

15. Exit the container.

```
root@zombie-<pod-suffix>:/# exit
```

16. Edit the zombie deployment file:

```
student@bchd:~$ vim zombie.yaml
```

17. Within the file, change the NGINX version from 1.18.0 into 1.19.6. The file below has comments around the change you should make.

Click here to view the contents of **zombie-roll-1196.yaml**

18. Save and exit with `:wq`

19. Now apply that change and see what the below pane shows as your rollout is taking effect.

```
student@bchd:~$ kubectl apply -f zombie.yaml

deployment.apps/zombie configured
```

20. Answer the following question:

    - **What happened?**
        - *Kubernetes created a new replica set containing a pod with the updated image. For each 'new pod' it created, it then terminated an 'old pod'. It repeated this process in a controlled manner until all the pods within the Deployment were running the new image.*

21. The following command will provide a history of rollouts.

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         deployed ver. 1.18.0
2         deployed ver. 1.18.0
```

22. Uh-oh. It looks like our **CHANGE-CAUSE** field didn't reset to `<none>`. That's because the `apply` command will compare the version of the configuration that you're pushing with the previous version and apply the changes you've made, without overwriting any automated changes to properties you haven't

specified. More on how the apply operation works with updating Kubernetes resources is available here: https://kubernetes.io/docs/concepts/cluster-administration/manage-deployment/#in-place-updates-of-resources

23. Manually set an annotation on the **CHANGE-CAUSE** field. *NOTE: Setting this annotation could be part of an automated process*

```
student@bchd:~$ kubectl annotate deployment zombie kubernetes.io/change-cause="ver. 1.19.6"

deployment.apps/zombie annotated
```

24. The following command will provide a history of rollouts. The new comment should be present.

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         deployed ver. 1.18.0
2         ver. 1.19.6
```

25. Get the pods again. Did the names of the pods change when you did a rollout? Select one to use for the next step.

```
student@bchd:~$ kubectl get pods

NAME                      READY   STATUS    RESTARTS   AGE
zombie-678645bf77-4dlkz   1/1     Running   0          7m46s
zombie-678645bf77-kf8jv   1/1     Running   0          8m26s
zombie-678645bf77-w658m   1/1     Running   0          8m14s
```

26. Now check on the pods and the containers within them. Is the NGINX version changed? You will notice that the pods have changed IDs. This is no mistake as it proves the rollout has worked.

```
student@bchd:~$ kubectl exec -it zombie-<replace-with-zombie-pod-id> -- bash
```

27. Once you are inside your container, check the version of NGINX that is running. If our upgrade worked, it should now read 1.19.6.

```
root@zombie-<pod-suffix>:/# nginx -v

nginx version: nginx/1.19.6
```

28. Exit the container.

```
root@zombie-<pod-suffix>:/# exit
```

29. Once again, edit the zombie deployment file:

```
student@bchd:~$ vim ~/zombie.yaml
```

30. Edit the manifest to rollout a 3rd update. This time, use the latest image available. See the comments for the latest edit.

Click here to view the contents of **zombie-roll-num3.yaml**

31. Save and exit with `:wq`

32. Now apply that change and see what the below pane shows as your rollout is taking effect. *This time we'll include the* `--record` *flag* which should auto update our **CHANGE-CAUSE** field.

```
student@bchd:~$ kubectl apply -f zombie.yaml --record

deployment.apps/zombie configured
```

> STOP! Read this. On the CKA/CKAD exam, you will be asked to ***'Record the change.'*** This is what they're looking for. However, as you will notice in this lab, the ***--record*** flag has been deprecated. This mean that it is scheduled to be removed. It has been in this state for about 5 years, and no alternative has yet to be made available. Therefore, the ***--record*** flag is the appropriate resolution for this task. If you would like to look at what a five year old, controversial GitHub ticket looks like, check out the incredible ***--record*** drama unfolding on the Kubernetes GitHub. https://github.com/kubernetes/kubernetes/issues/40422

33. The following command will provide a history of rollouts.

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         deployed ver. 1.18.0
2         ver. 1.19.6
3         kubectl apply --filename=zombie.yaml --record=true
```

34. At least this time Kubernetes didn't recopy the **CHANGE-CAUSE** field! Once again, manually set an annotation on the **CHANGE-CAUSE** field. *NOTE: Setting this annotation could be part of an automated process*

```
student@bchd:~$ kubectl annotate deployment zombie kubernetes.io/change-cause="deployed latest stable release"

deployment.apps/zombie annotated
```

35. Ensure all of your updates are clearly annotated.

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         deployed ver. 1.18.0
2         ver. 1.19.6
3         deployed latest stable release
```

ROLLBACKS

36. Now we want to actually do a rollback. We will walk through similar steps to see if the rollback works.

37. Take a quick look at your rollout history details concerning, "Revision #1". The command you are about to issue is strictly informational (it will not make any changes).

```
student@bchd:~$ kubectl rollout history deploy zombie --revision=1

deployment.apps/zombie with revision #1
Pod Template:
  Labels:       app=nginx
        pod-template-hash=5bf87f5f59
  Annotations:  kubernetes.io/change-cause: deployed ver. 1.18.0
  Containers:
   nginx:
    Image:      nginx:1.18.0
    Port:       80/TCP
    Host Port:  0/TCP
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
```

38. Take a quick look at your rollout history details concerning, "Revision #2." The command you are about to issue is strictly informational (it will not make any changes).

```
student@bchd:~$ kubectl rollout history deploy zombie --revision=2

deployment.apps/zombie with revision #2
Pod Template:
  Labels:       app=nginx
        pod-template-hash=678645bf77
  Annotations:  kubernetes.io/change-cause: ver. 1.19.6
  Containers:
   nginx:
    Image:      nginx:1.19.6
    Port:       80/TCP
    Host Port:  0/TCP
    Environment:        <none>
    Mounts:     <none>
  Volumes:      <none>
```

39. Check the status of the rollouts. We completed our rollout several steps ago, so it should show a completed state. Remember, this upgraded our nginx images to the latest stable version of nginx.

```
student@bchd:~$ kubectl rollout status deploy zombie

deployment "zombie" successfully rolled out
```

40. Now attempt to undo a rollout via a rollback. The 'undo' command undoes the most recent update.

```
student@bchd:~$ kubectl rollout undo deploy/zombie

deployment.apps/zombie rolled back
```

41. Once the rollback has completed, check on the rollback history.

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         deployed ver. 1.18.0
3         deployed latest stable release
4         ver. 1.19.6
```

42. Interesting! The revision is now `4`, and the **CHANGE-CAUSE** field indicates we should be on `ver 1.19.6`. Let's check that out.

43. Get the pods again.

```
student@bchd:~$ kubectl get pods

NAME                      READY   STATUS    RESTARTS   AGE
zombie-678645bf77-crk8b   1/1     Running   0          87s
zombie-678645bf77-d8vrr   1/1     Running   0          85s
zombie-678645bf77-gc8ks   1/1     Running   0          89s
```

44. Now check on the pods and the containers within them. Is the NGINX version changed? You will notice that the pods have changed IDs. This is no mistake as we hope that the rollout has worked.

```
student@bchd:~$ kubectl exec -it zombie-<replace-with-zombie-pod-id> -- bash
```

45. Check the NGINX version. If it actually rolled back, it should now reflect 1.19.6, the version *before* we moved to the stable version.

```
root@zombie-<pod-suffix>:/# nginx -v

nginx version: nginx/1.19.6
```

46. Exit the container.

```
root@zombie-<pod-suffix>:/# exit
```

47. The rollout status should also show that it was successful.

```
student@bchd:~$ kubectl rollout status deploy zombie

deployment "zombie" successfully rolled out
```

48. Kubernetes always increases the **REVISION** value, but notice that the **CHANGE-CAUSE** was 'moved' to this new revision! Cool! Taking a little extra time to organize ourselves is *REALLY* paying off!!

```
student@bchd:~$ kubectl rollout history deploy zombie

deployment.apps/zombie
REVISION  CHANGE-CAUSE
1         deployed ver. 1.18.0
3         deployed latest stable release
4         ver. 1.19.6
```

49. **CHALLENGE 01 (OPTIONAL) -** To rollback to a specific revision on "zombie", you would issue `kubectl rollout undo deploy/zombie --to-revision=n`, where 'n' is the revision you wish to select. Try to rollback to the oldest revision listed available to you.

50. *CHALLENGE 01 (SOLUTION) - Issue the following commands: kubectl rollout history deploy zombie*
kubectl rollout undo deploy/zombie --to-revision=n # be sure to replace n with a revision number

51. When you're done with this lab, close your 'extra' tmux window by typing `exit`.

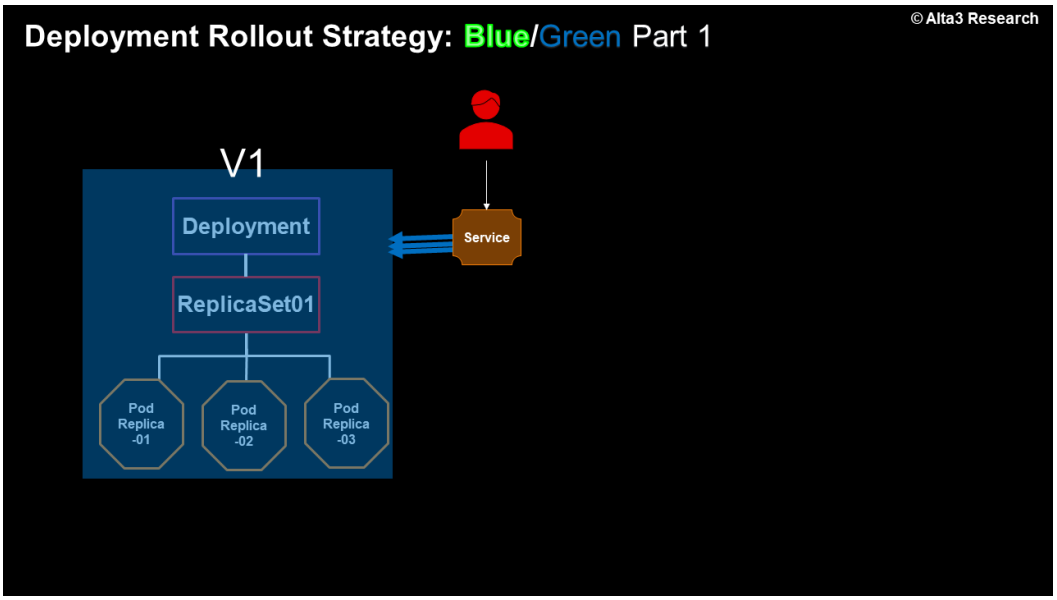
52. To stop the **watch** press `CTRL+C`

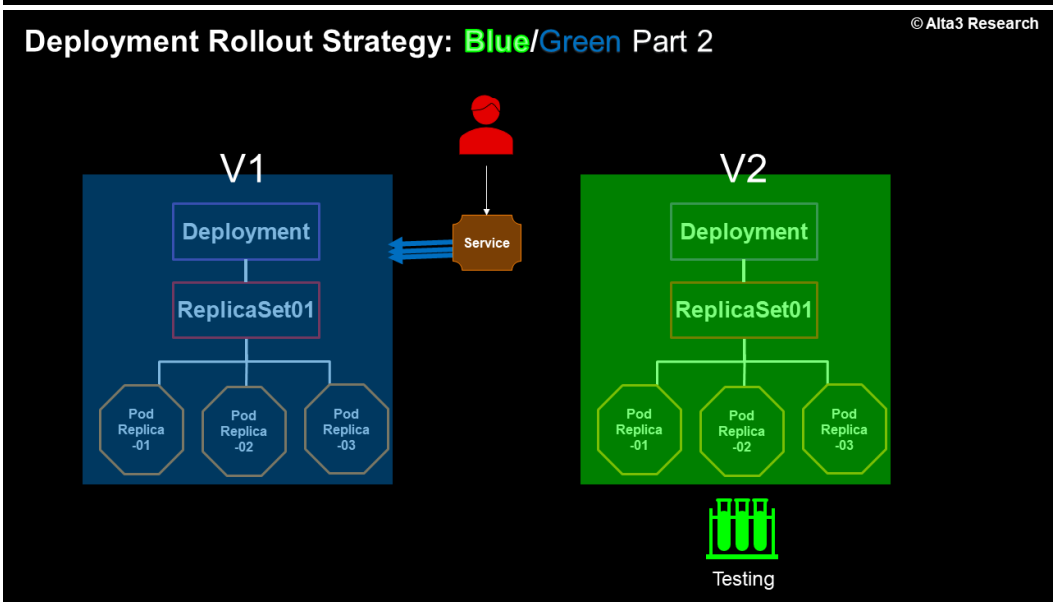
53. Delete your "zombies".

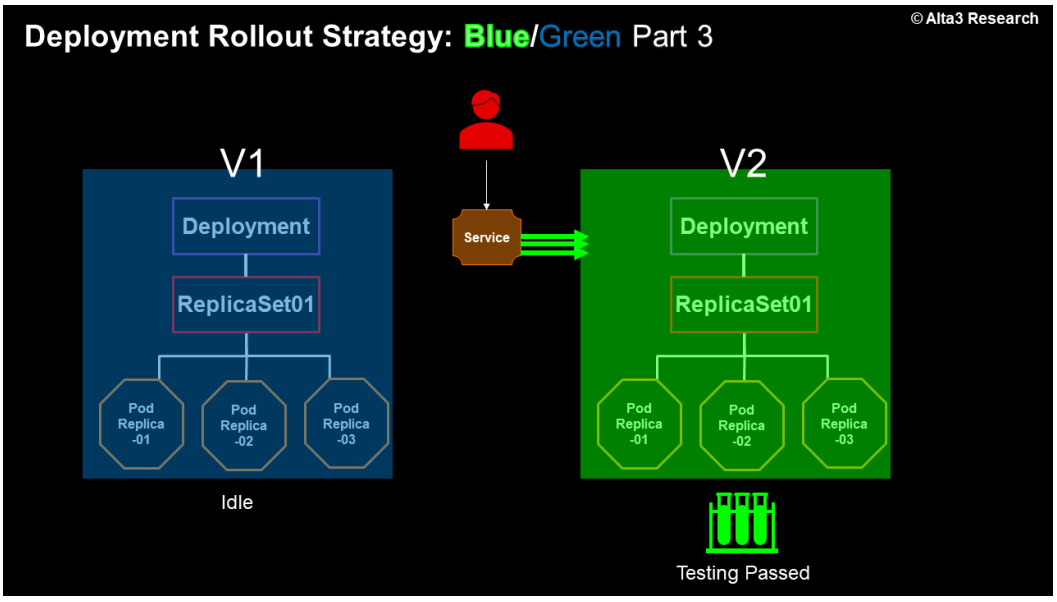
```
student@bchd:~$ kubectl delete deploy zombie

deployment.apps "zombie" deleted
```

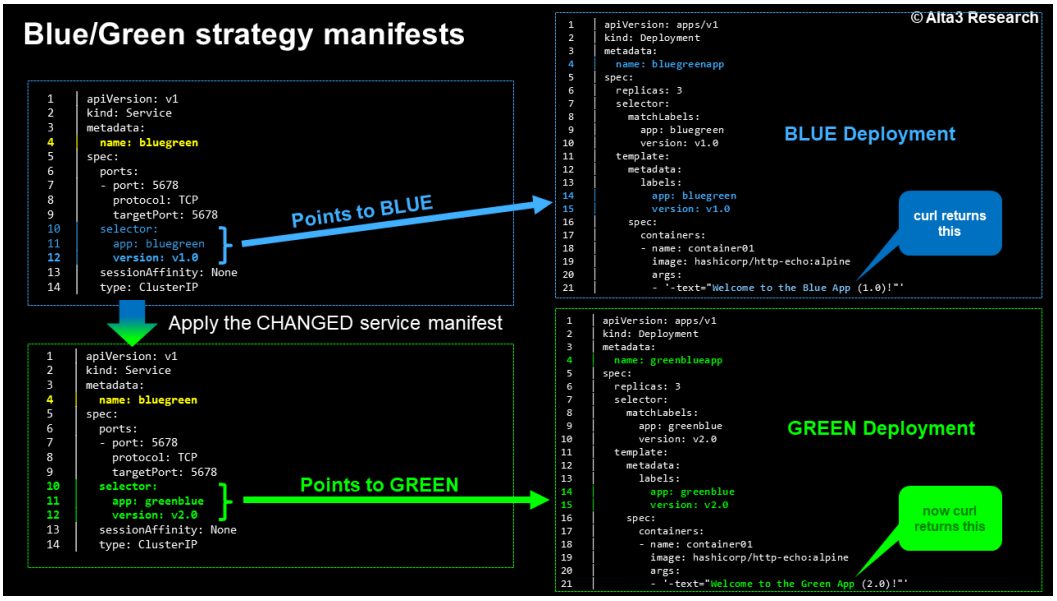
54. That's it for this lab!

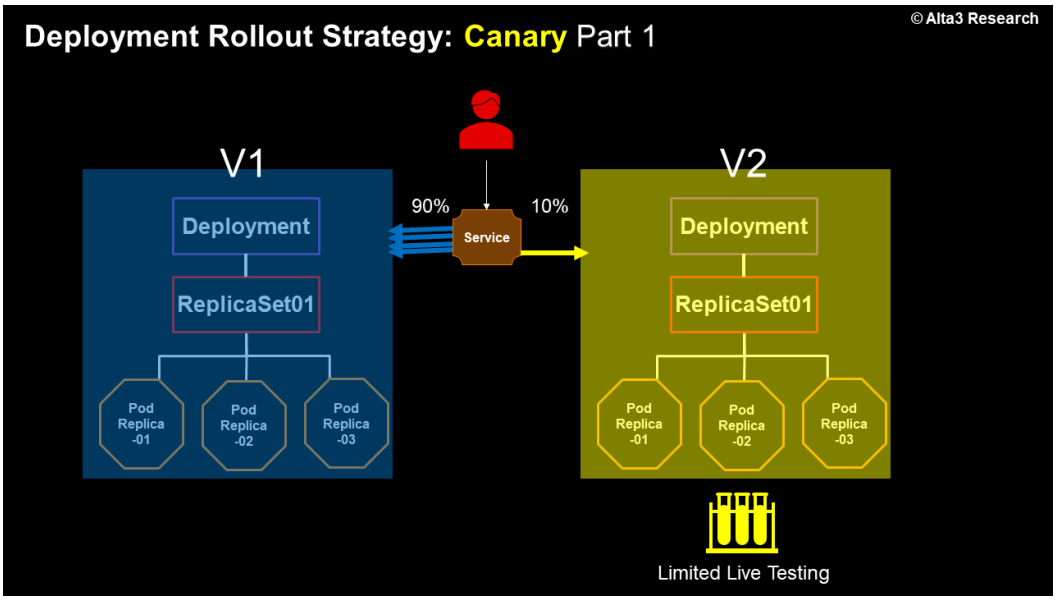
49. Blue/Green and Canary Deployment Strategies



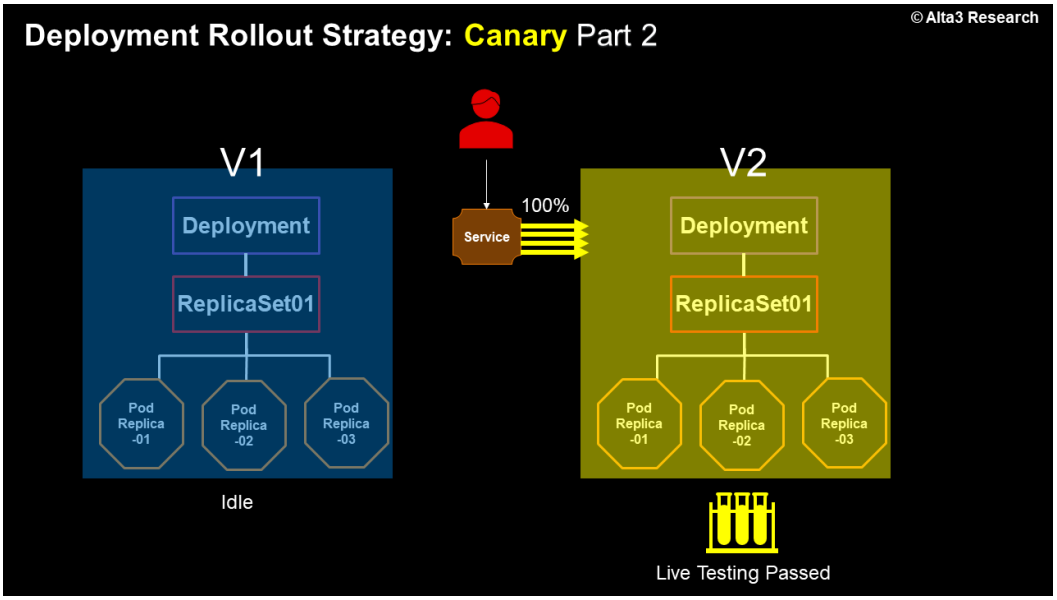


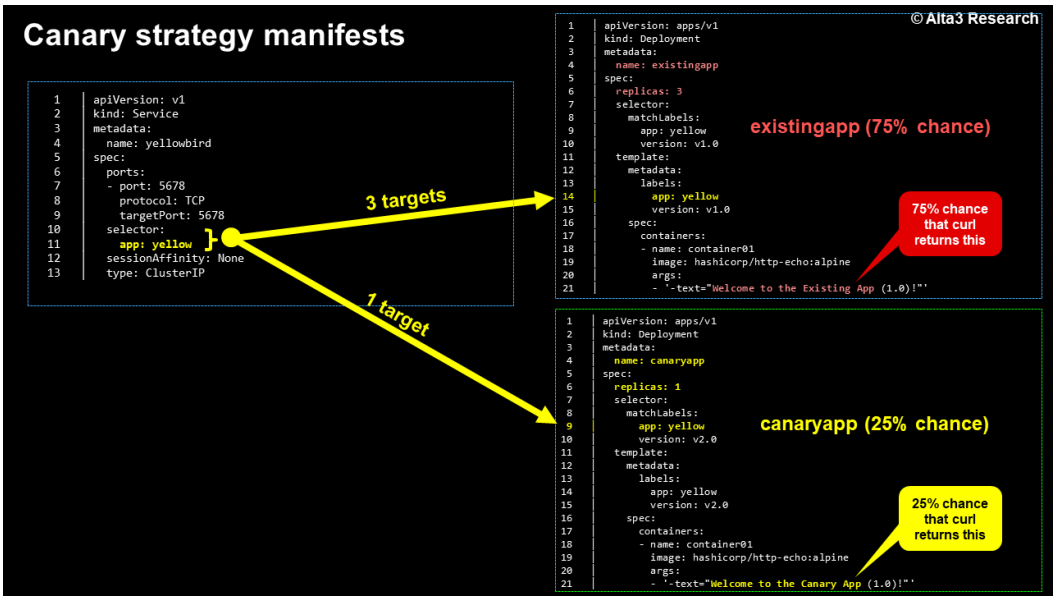






Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com





50. Advanced Deployment Strategies

Along with the **Recreate** and **RollingUpdate** strategies, it is also worth exploring two more advanced deployment strategies: **Blue/Green** and **Canary** deployments.

Blue/Green Deployments

1. First, we will want to create a deployment.

```
student@bchd:~$ vim blue.yml
```

Click here to view the contents of **blue.yml**

2. Instantiate the deployment.

```
student@bchd:~$ kubectl apply -f blue.yml
```

3. Verify that the deployment has been created as expected.

```
student@bchd:~$ kubectl get deploy

NAME           READY   UP-TO-DATE   AVAILABLE   AGE
bluegreenapp   3/3     3            3           6s
```

4. Next we will create a ClusterIP service for access to our *blue* deployment.

```
student@bchd:~$ vim bluegreensvc.yml
```

Click here to view the contents of **bluegreensvc.yml**

5. Start the bluegreen service.

```
student@bchd:~$ kubectl apply -f bluegreensvc.yml

service/bluegreen created
```

6. Now inspect the endpoints that the service is keeping track of.

```
student@bchd:~$ kubectl get endpoints bluegreen -o yaml | grep name:

name: bluegreen
    name: bluegreenapp-67886f8667-8f76h
    name: bluegreenapp-67886f8667-hm57x
    name: bluegreenapp-67886f8667-9lvlm
```

7. Now, let's create a new version of this app, the **Green** version.

```
student@bchd:~$ vim green.yml
```

Click here to view the contents of **green.yml**

8. Start up our second version of this deployment, running parallel to our first.

```
student@bchd:~$ kubectl apply -f green.yml
```

9. Verify that the new deployment has been created as expected.

```
student@bchd:~$ kubectl get deploy

NAME           READY   UP-TO-DATE   AVAILABLE   AGE
bluegreenapp   3/3     3            3           84s
greenblueapp   3/3     3            3           6s
```

10. Let's update the service so that it is looking at our other deployment. Make sure it looks exactly like the below.

```
student@bchd:~$ vim bluegreensvc.yml
```

Click here to view the contents of **greenbluesvc.yml**

11. Apply the changes to our service in order to update our selectors.

```
student@bchd:~$ kubectl apply -f bluegreensvc.yml
```

12. Let's once again see what endpoints our service is selecting now.

```
student@bchd:~$ kubectl get endpoints bluegreen -o yaml | grep name:

name: bluegreen
    name: greenblueapp-b7595fc7b-5wkgk
    name: greenblueapp-b7595fc7b-wfr4m
    name: greenblueapp-b7595fc7b-dfzjp
```

13. Excellent, let's clean up our deployments now.

```
student@bchd:~$ kubectl delete deploy --all
```

Canary Deployments

Canary Deployments allow us to once again have two parallel instances of the deployments running, but only direct a portion of the traffic coming into our service towards the new version of the deployment. Then, when the new **Canary** version of the deployment has been adequately tested, you can simply increase the percent of traffic going to the newer version.

14. Let's make another deployment called existingapp.

```
student@bchd:~$ vim existingapp.yml
```

Click here to view the contents of **existingapp.yml**

15. Let's also make a version 2 of our deployment.

```
student@bchd:~$ vim canaryapp.yml
```

Click here to view the contents of **canaryapp.yml**

16. Start your deployments!

```
student@bchd:~$ kubectl apply -f existingapp.yml -f canaryapp.yml
```

17. Verify that both deployments are up and have all of their expected pods in a ready state.

```
student@bchd:~$ kubectl get deploy
```

18. Now we need to create a service that will be able to direct traffic to both of these deployments. Notice that this time, our service **will not select by version**.

```
student@bchd:~$ vim yellowbird.yml
```

Click here to view the contents of **yellowbird.yml**

19. Start your service!

```
student@bchd:~$ kubectl apply -f yellowbird.yml
```

20. Check out the service.

```
student@bchd:~$ kubectl describe svc yellowbird

Name:              yellowbird
Namespace:         default
Labels:            <none>
Annotations:       Selector:  app=yellow
Type:              ClusterIP
IP:                172.16.3.218
Port:              <unset>  5678/TCP
TargetPort:        5678/TCP
Endpoints:         192.168.139.82:5678,192.168.247.14:5678,192.168.84.140:5678 + 1 more...
Session Affinity:  None
Events:            <none>
```

21. Now inspect the endpoints that the service is tracking.

```
student@bchd:~$ kubectl get endpoints yellowbird -o yaml | grep name:
```

```
name: yellowbird
  name: existingapp-556878c678-pdr5g
  name: existingapp-556878c678-bvhbf
  name: existingapp-556878c678-jp9zr
  name: canaryapp-7948d6dcc5-22ccn
```

Here we see that 25% of our traffic will be flowing to the *canaryapp* version, while 75% remains flowing to the older *existingapp* deployment.

22. Let's create a fun script to actually see how our data will be split between the **existingapp**, and the new **canaryapp**!

```
student@bchd:~$ vim testpine.yml
```

Click here to view the contents of **testpine.yml**

To save and quit out of vim, type `:wq` and press **ENTER**.

23. Now, create our new testing pod!

```
student@bchd:~$ kubectl apply -f testpine.yml

pod/testpine created
```

24. Run the following commands as a single entry.

```
CLUSTERIP=`kubectl get svc yellowbird --template '{{.spec.clusterIP}}'`
kubectl exec -it testpine -- apk update
kubectl exec -it testpine -- apk upgrade
kubectl exec -it testpine -- apk add curl
```

25. Use testpine as the source Pod to test connectivity to the yellowbird service. You will see either Canary or Existingapp response but not both at the same time.

```
student@bchd~$ kubectl exec -it testpine -- curl http://$CLUSTERIP:5678

"Welcome to the Canary App (1.0)!"
            OR
"Welcome to the Existing App (1.0)!"
```

26. Let's repeat that previous step 20 times. Do you notice the destination changes randomly?

```
student@bchd~$ for i in {1..20}; do kubectl exec -it testpine -- curl http://$CLUSTERIP:5678; done

"Welcome to the Canary App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Canary App (1.0)!"
"Welcome to the Canary App (1.0)!"
"Welcome to the Canary App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Canary App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Canary App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Canary App (1.0)!"
"Welcome to the Existing App (1.0)!"
"Welcome to the Existing App (1.0)!"
```

27. Now that we are ready to move forward with our next version of our app, we will simply cause all of the traffic to flow to the new version through our built in ability to scale.

28. We will first scale up the new version, and then scale down the old version of the deployment to zero.

```
student@bchd:~$ kubectl scale deploy canaryapp --replicas 4 && kubectl scale deploy existingapp --replicas 0
```

29. Let's see how that has affected which endpoints our service is keeping track of.

```
student@bchd:~$ kubectl get endpoints yellowbird -o yaml | grep name:

name: yellowbird
    name: canaryapp-7948d6dcc5-mdmrh
    name: canaryapp-7948d6dcc5-8swth
    name: canaryapp-7948d6dcc5-gdlld
    name: canaryapp-7948d6dcc5-22ccn
```

We now have 100% of our traffic flowing to the newly deployed v2.0!

30. Run the curl test again and spot the difference!

```
student@bchd~$ for i in {1..20}; do kubectl exec -it testpine -- curl http://$CLUSTERIP:5678; done
```

31. Excellent, let's clean up after ourselves now.

```
student@bchd:~$ kubectl delete deploy --all
```

51. Deployments Scaling

52. Horizontal Scaling with kubectl scale

Lab Objective

- Use the kubectl *scale* command to scale up to a certain amount of replicas.

Using kubectl scale you can achieve imperative scaling. Imperative commands are useful for demonstrations and quick reactions to emergency situations; it is important to also update any text-file configurations to match the number of replicas that you set via the imperative scale command.

Here are some additional scaling resources for you to check out:

- https://medium.com/bitnami-perspectives/imperative-declarative-and-a-few-kubectl-tricks-9d6deabdde
- https://kubernetes.io/docs/concepts/overview/object-management-kubectl/declarative-config/
- https://kubernetes.io/docs/concepts/overview/object-management-kubectl/imperative-command/

Procedure

1. Setup your lab environment:

```
student@bchd:~$ setup horizontal-scaling-with-kubectl-scale
```

2. We need to create a deployment first so we can imperatively scale it out.

```
student@bchd:~$ batcat ~/mycode/yaml/sise-deploy.yaml
```

Click here to view the contents of **sise-deploy.yaml**

3. Create the deployment now.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/sise-deploy.yaml

deployment.apps/sise-deploy created
```

4. Describe the Deployment, grepping for replicas. Also get the pods to make sure the correct number of replicas are running. (Keep that number in mind, as we will attempt to change that soon)

```
student@bchd:~$ kubectl describe deployments sise-deploy | grep Replicas

Replicas:               2 desired | 2 updated | 2 total | 2 available | 0 unavailable
  Available      True    MinimumReplicasAvailable

student@bchd:~$ kubectl get pods

NAME                           READY   STATUS    RESTARTS   AGE
sise-deploy-7566bd957d-ffssg   1/1     Running   0          2m48s
sise-deploy-7566bd957d-q6rd5   1/1     Running   0          2m48s
```

5. Scale out the number of replicas for the `sise` pod.

```
student@bchd:~$ kubectl scale deployment sise-deploy --replicas=3

deployment.apps/sise-deploy scaled
```

6. Get the pods. There are three of them now. Notice that the newest one is only a few seconds old.

```
student@bchd:~$ kubectl get pods

NAME                           READY   STATUS    RESTARTS   AGE
sise-deploy-6f56b9b46c-4kmjk   1/1     Running   0          2m43s
sise-deploy-6f56b9b46c-bxgx5   1/1     Running   0          2m43s
sise-deploy-6f56b9b46c-s2pxm   1/1     Running   0          3s
```

7. If you are quick enough, you may see the Status of your new pods changing.

```
student@bchd:~$ kubectl describe deployments sise-deploy
```

```
Name:                   sise-deploy
Namespace:              default
CreationTimestamp:      Sun, 22 May 2022 19:48:13 +0000
Labels:                 <none>
Annotations:            deployment.kubernetes.io/revision: 1
Selector:               app=sise
Replicas:               3 desired | 3 updated | 3 total | 3 available | 0 unavailable
StrategyType:           RollingUpdate
MinReadySeconds:        0
RollingUpdateStrategy:  25% max unavailable, 25% max surge
Pod Template:
  Labels:  app=sise
  Containers:
   sise:
    Image:      mhausenblas/simpleservice:0.5.0
    Port:       9876/TCP
    Host Port:  0/TCP
    Environment:
      SIMPLE_SERVICE_VERSION:  1.0
    Mounts:                    <none>
  Volumes:                     <none>
Conditions:
  Type           Status  Reason
  ----           ------  ------
  Progressing    True    NewReplicaSetAvailable
  Available      True    MinimumReplicasAvailable
OldReplicaSets:  <none>
NewReplicaSet:   sise-deploy-6f56b9b46c (3/3 replicas created)
Events:
  Type    Reason             Age    From                   Message
  ----    ------             ----   ----                   -------
  Normal  ScalingReplicaSet  5m34s  deployment-controller  Scaled up replica set sise-deploy-6f56b9b46c to 2
  Normal  ScalingReplicaSet  2m54s  deployment-controller  Scaled up replica set sise-deploy-6f56b9b46c to 3
```

Declarative Scaling with *kubectl apply*

> Now you will scale the ReplicaSet out to another number of replicas. Then we will be able to see the difference between imperative and declarative scaling. In a declarative world, we make changes by editing the actual configuration file in version control and then applying those changes to the cluster.

8. Make a copy of the current sise-deploy.yaml

```
student@bchd:~$ cp ~/mycode/yaml/sise-deploy.yaml  ~/mycode/yaml/sise-deploy-5.yaml
```

9. Take a look at the manifest we used to create the Deployment. Find the `spec:` section with `replicas` underneath and change the value, one up or one down from what it was. An example of what to search for is below.

```
student@bchd:~$ vim ~/mycode/yaml/sise-deploy-5.yaml

...
spec:
  replicas: 5  # change this value to something OTHER than what it was...
...
```

> Make sure to `:wq`!

10. Submit the changes to the API Server.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/sise-deploy-5.yaml
```

> If you were adding to the replica count or you were reducing the amount of replicas, you will respectively either see *ContainerCreating* or *Terminating* if you do a `kubectl get pods` quickly enough.

11. Now that the ReplicaSet has been updated and submitted to the API Server, the Replication Controller will scan in these changes and recognize that the number of desired Pods has a new value and it needs to take action in order for that desired value to become true. If you did not complete the above, run the `kubectl get rs` command to see the updated number of pods via the ReplicaSet.

```
student@bchd:~$ kubectl get rs

NAME                    DESIRED   CURRENT   READY   AGE
sise-deploy-6f56b9b46c   5         5         5       9m10s
```

12. Let's run through what we just did a second time, only now replace the `simpleservice` image with the tagged version of the website (`webby`).

13. Begin with your blank **webby-deploy.yaml** file and edit it to match below.

```
student@bchd:~$ vim ~/mycode/yaml/webby-deploy.yaml
```

Click here to view the contents of **webby-deploy.yaml**

14. When you are finished, it should look like the below.

Click here to view the contents of **webby-deploy-filled-out.yaml**

15. Create the Deployment.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/webby-deploy.yaml
```

16. Imperatively scale the Deployment.

```
student@bchd:~$ kubectl scale deployment webservice --replicas=2
```

> Verify with a `kubectl get pods` command

17. Declaratively scale the Deployment. Open your Deployment Manifest and edit the `spec.replicas` value to `7`.

```
student@bchd:~$ vim ~/mycode/yaml/webby-deploy.yaml

...
spec:
  replicas: 7
...
```

18. Apply the new Deployment configuration.

```
student@bchd:~$ kubectl apply -f ~/mycode/yaml/webby-deploy.yaml
```

19. Once you have finished inspecting the new Deployment, let's delete all of our Deployments.

```
student@bchd:~$ kubectl delete deploy --all
```

53. Jobs Cronjobs

Job

© Alta3 Research 1

K8s job

- Creates one or more Pods that run a batch process to completion.
- Ensures that a specified number of Pods run in parallel.
- Deleting a Job will cleanup the Pods it created.

Job Spawned Pods

- Unless they are stopped, they keep running!

Job Patterns

© Alta3 Research 2

A K8s job is specified by

- **Completions**
- **Parallelism (simultaneously running Pods)**

For instance:

You must calculate 2,000 digital certificates quickly, so which pattern will you use?

Given: A job will create ONE KEY!

A. Enter a job command 2000 times

B. Enter a single job command that repeats 2000 times, creating one key at a time.

C. Enter a single job command that creates 500 Pods, each creating one key, repeating four times, creating 500 keys at a time.

D. Enter a single job command that launches 2000 Pods, each creating 1 key

Parallelism

© Alta3 Research 3

Do five jobs at a time

Complete 10 jobs total

```
apiVersion: batch/v1
kind: Job
metadata:
  name: parallel
  labels:
    chapter: jobs
spec:
  parallelism: 5
  completions: 10
  template:
    metadata:
      labels:
        chapter: jobs
    spec:
      containers:
      - name: kuard
        image: kuard:latest
        imagePullPolicy: Always
        args:
        - "--keygen-enable"
        - "--keygen-exit-on-complete"
        - "--keygen-num-to-gen=10"
      restartPolicy: OnFailure
```

54. Kubernetes Jobs and CronJobs

CKAD Objective

- Understand Jobs and Cron Jobs

Lab Objectives

1. Run a simple echo job
2. Convert your echo job to a cronjob

Oftentimes during our day, it becomes necessary to get a job done quickly. One approach is to slice the job into smaller tasks, then assign a team of workers to do it all at the same time. This is what a Kubernetes job does for us.

Procedure

1. Create the echo job and then apply the yaml file.

```
student@bchd:~$ vim echocomplete.yaml

apiVersion: batch/v1
kind: Job
metadata:
  name: echocomplete
spec:
  parallelism: 5
  completions: 20
  template:
    metadata:
      name: echocomplete
    spec:
      containers:
      - name: echo
        image: centos:7
        command: ["echo", "This text here means success!"]
      restartPolicy: Never

student@bchd:~$ kubectl apply -f echocomplete.yaml
```

2. Watch the job in a new TMUX pane. Shortly, the job will show that it has completed successfully. Nothing else to see. `Ctrl c` to stop the watch.

```
student@bchd:~$ watch kubectl get jobs

NAME                     COMPLETIONS  DURATION  AGE
echocomplete                20/20          12s       10s
```

3. When the job initially runs, it spawns a pod name, then runs under that name, and then it dies. Sad but true! However it leaves its logs behind which you can collect and examine. First get a list of pods that were created. Notice how many pods that appear. Twenty pods should pop up from this command because we asked this job to be completed twenty times.

```
student@bchd:~$ kubectl get pods
```

```
NAME                 READY   STATUS      RESTARTS   AGE
echocomplete-9pmwg   0/1     Completed   0          4m2s
echocomplete-bjsk8   0/1     Completed   0          4m3s
echocomplete-cpd7z   0/1     Completed   0          3m59s
echocomplete-dx774   0/1     Completed   0          3m58s
echocomplete-f55ql   0/1     Completed   0          4m4s
echocomplete-fcsv6   0/1     Completed   0          4m4s
echocomplete-gz95q   0/1     Completed   0          4m7s
echocomplete-hwcwt   0/1     Completed   0          4m5s
echocomplete-k6t5k   0/1     Completed   0          3m59s
echocomplete-kkhwh   0/1     Completed   0          4m
echocomplete-mncm6   0/1     Completed   0          4m1s
echocomplete-pmndx   0/1     Completed   0          4m1s
echocomplete-r42sd   0/1     Completed   0          4m7s
echocomplete-rm8dj   0/1     Completed   0          4m7s
echocomplete-tltzg   0/1     Completed   0          4m4s
echocomplete-wrjnp   0/1     Completed   0          4m1s
echocomplete-x4j7r   0/1     Completed   0          4m7s
echocomplete-x9llf   0/1     Completed   0          4m7s
echocomplete-xxbwt   0/1     Completed   0          4m5s
echocomplete-xzxpc   0/1     Completed   0          4m3s
```

4. Test to see if the task completed successfully. Print the logs to see the standard output. You'll have to choose a specific pod from your list.

```
student@bchd:~$ kubectl logs <POD-NAME>

This text here means success!
```

5. Delete the job. Remember: when the Job is deleted so are the pods it created!

```
student@bchd:~$ kubectl delete job echocomplete

job.batch "echocomplete" deleted
```

6. When clicking on the link in the next step, hold the `Ctrl` key as you press the HERE link. This action will open the link in a new browser tab.

7. **Please read carefully! MANDATORY CHALLENGE:** Modify the YAML you just ran to run every minute, following the instructions here: https://kubernetes.io/docs/tasks/job/automated-tasks-with-cron-jobs/#creating-a-cron-job. This will be a "stare-and-compare" task for you to do. Simply look at the manifest in the documentation and make yours look similar to it. **You will need to change the manifest below! Please keep the name of the cron job as echocomplete**. To practice making CronJob schedules follow this link: https://crontab.guru/

```
student@bchd:~$ vim echocomplete.yaml

apiVersion: batch/v1
kind: Job
metadata:
  name: echocomplete
spec:
  parallelism: 5
  completions: 20
  template:
    metadata:
      name: echocomplete
    spec:
      containers:
      - name: echo
        image: centos:7
        command: ["echo", "This text here means success!"]
      restartPolicy: Never
```

8. Once you have edited this manifest to be a cronjob, run it.

```
student@bchd:~$ kubectl apply -f echocomplete.yaml

cronjob.batch/echocomplete created
```

9. Now, watch the jobs automatically start every minute.

```
student@bchd:~$ kubectl get pods -w
```

Watch this for 2 or 3 minutes to assure that the timing is correct before moving on.

10. Get the logs of one of the Job pods that has the status `Completed`. Remember that the syntax to do so is:

```
student@bchd:~$ kubectl logs <POD-NAME>
```

11. Delete your CronJob, or it will run continuously. Remember: when a CronJob is deleted so are the pods it created!

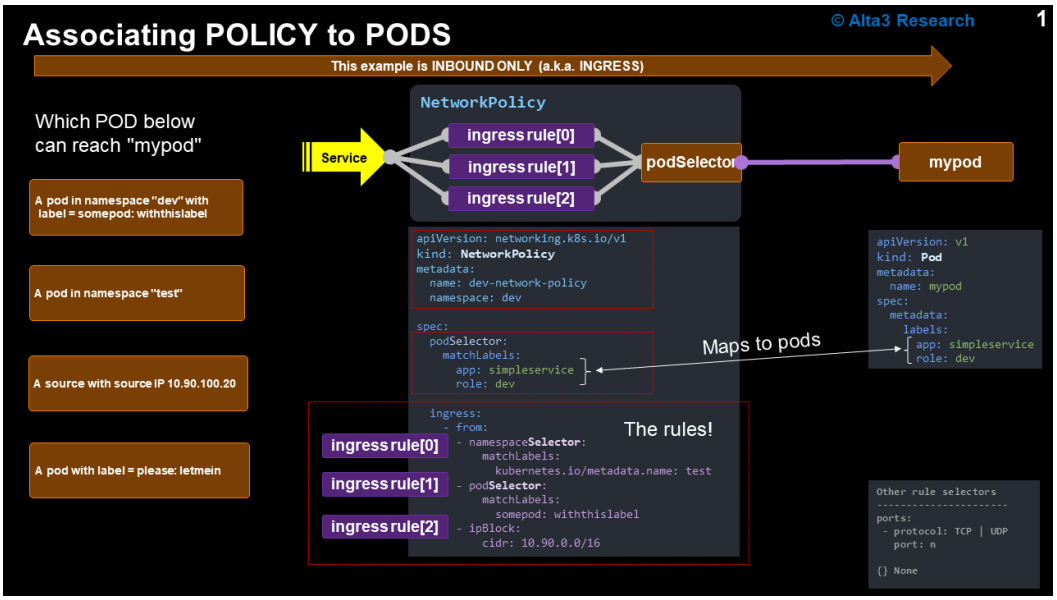
```
student@bchd:~$ kubectl delete cj echocomplete
```

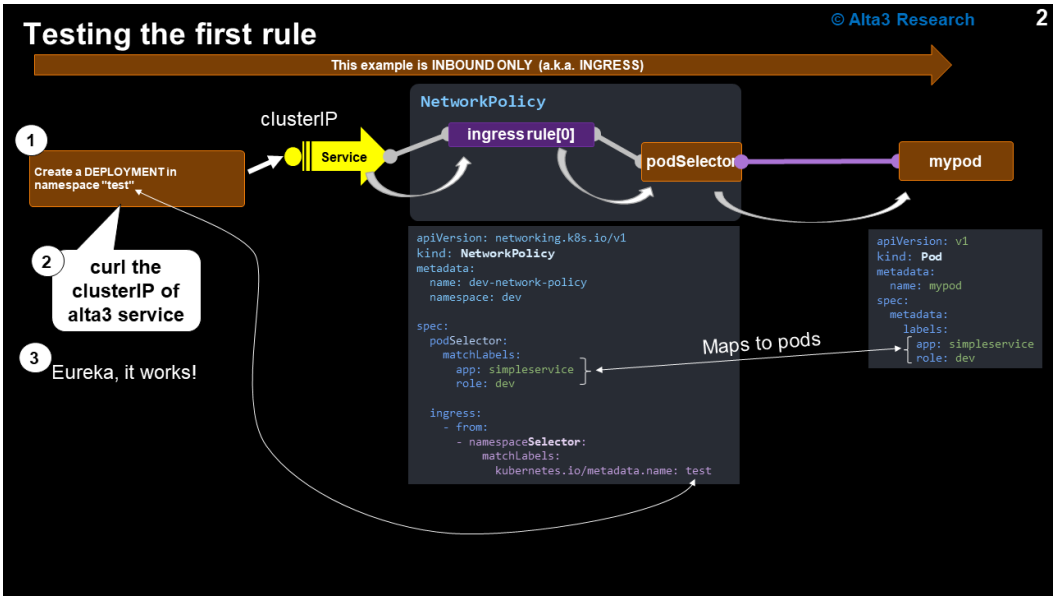
12. Nice *job*! Lab completed!

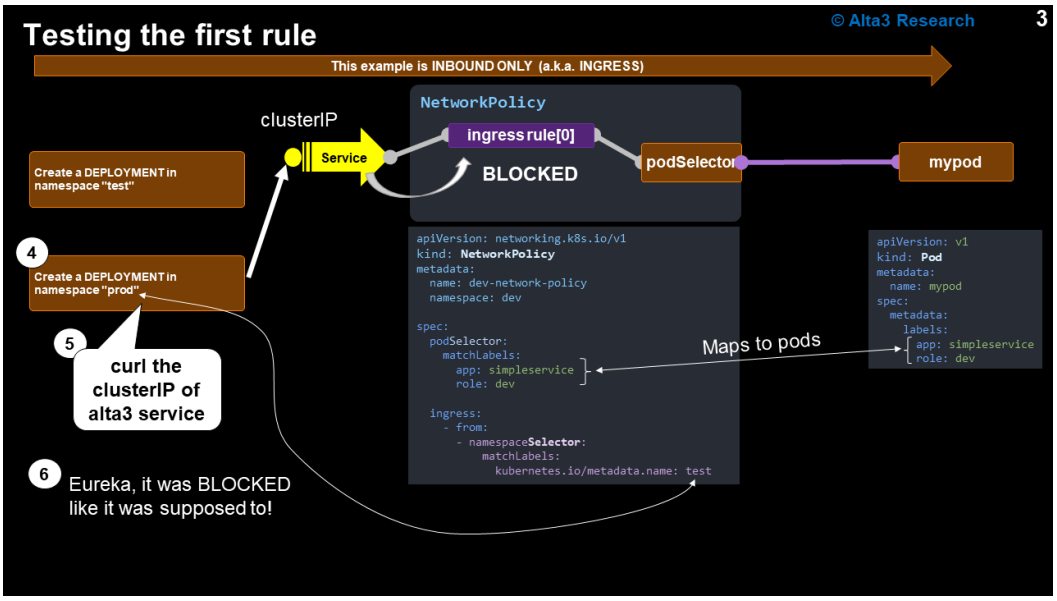
ANSWER *Here is the completed CronJob*.

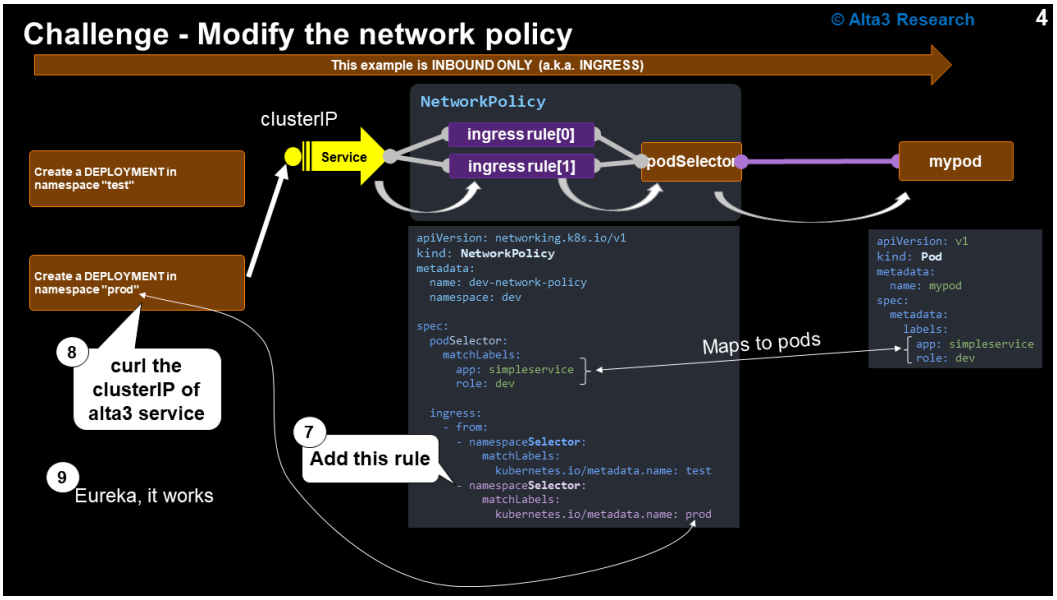
```
apiVersion: batch/v1
kind: CronJob
metadata:
  name: echocomplete
spec:
  schedule: "*/1 * * * *"
  jobTemplate:
    spec:
      parallelism: 5
      completions: 20
      template:
        spec:
          containers:
          - name: echo
            image: centos:7
            command: ["echo", "This text here means success!"]
          restartPolicy: Never
```

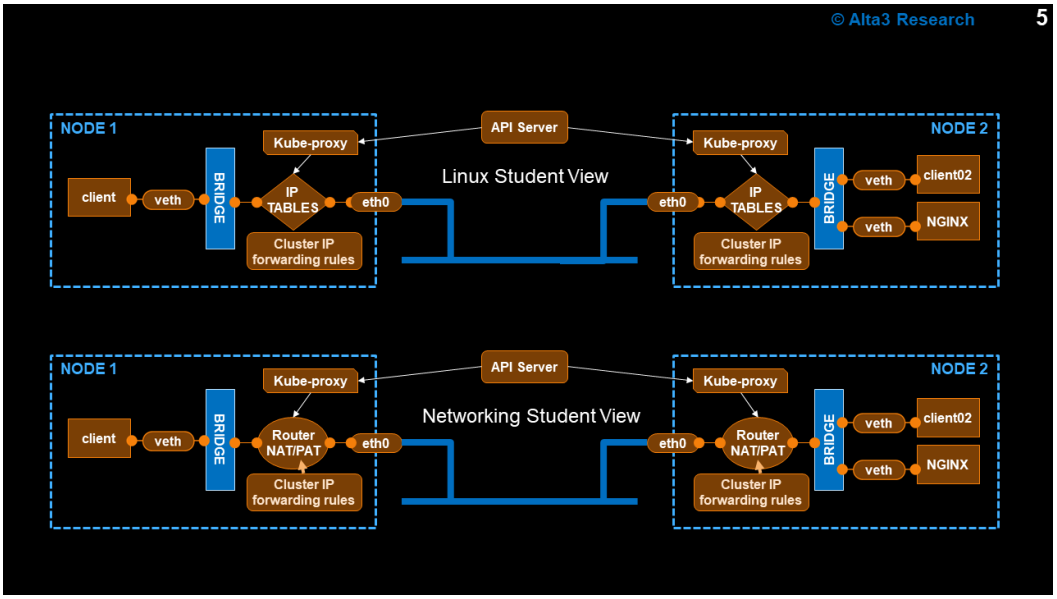
55. Network Policies

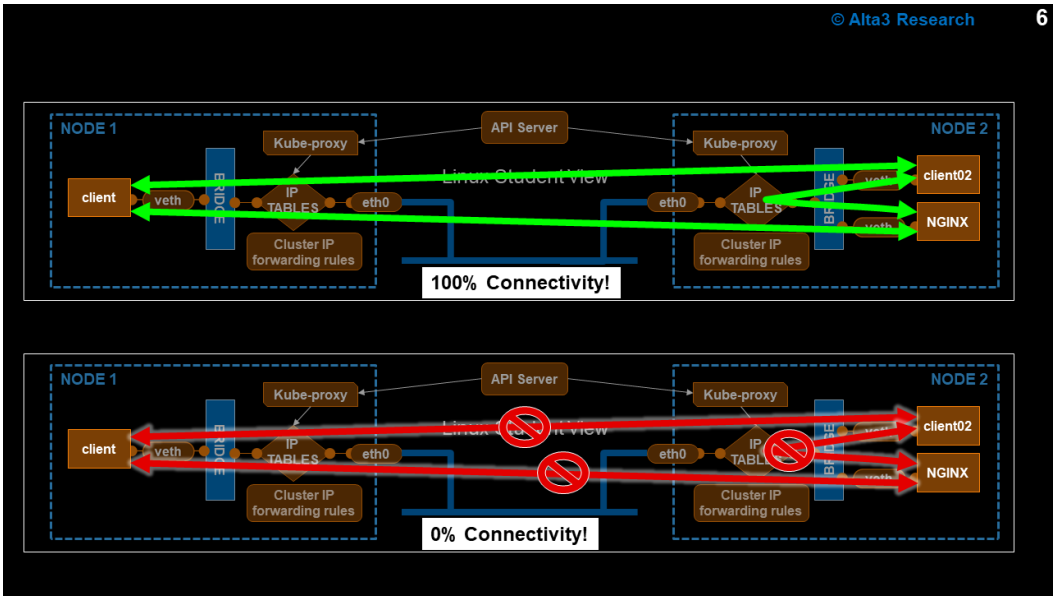


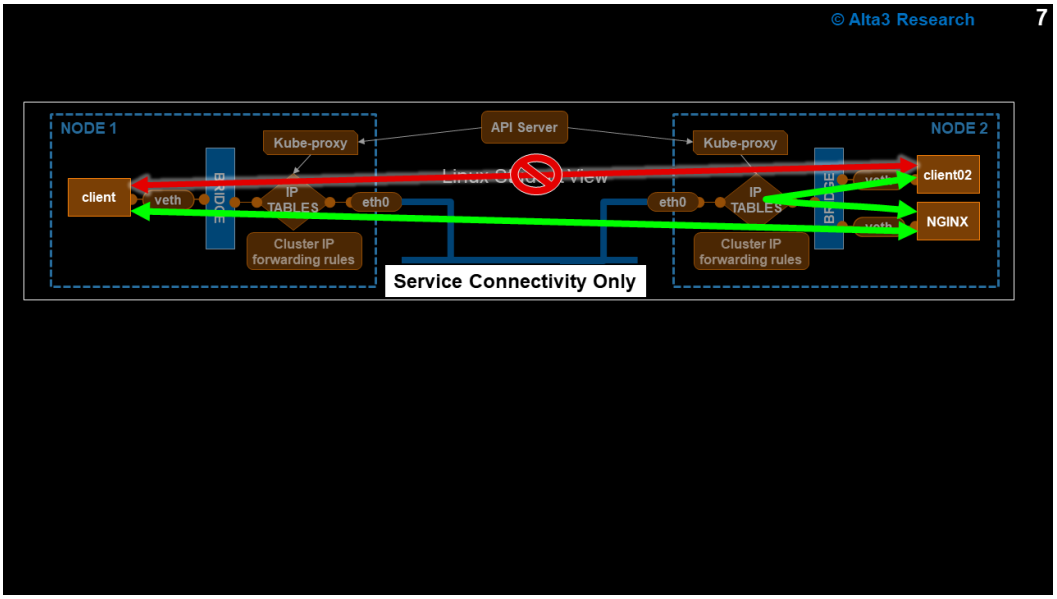












56. Namespace Network Policy

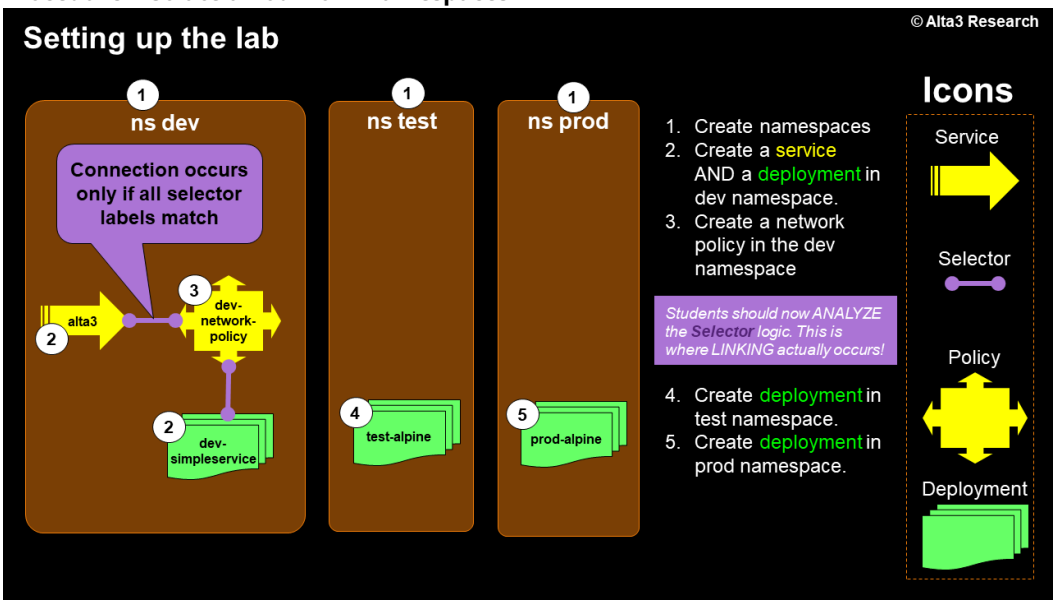
Objective

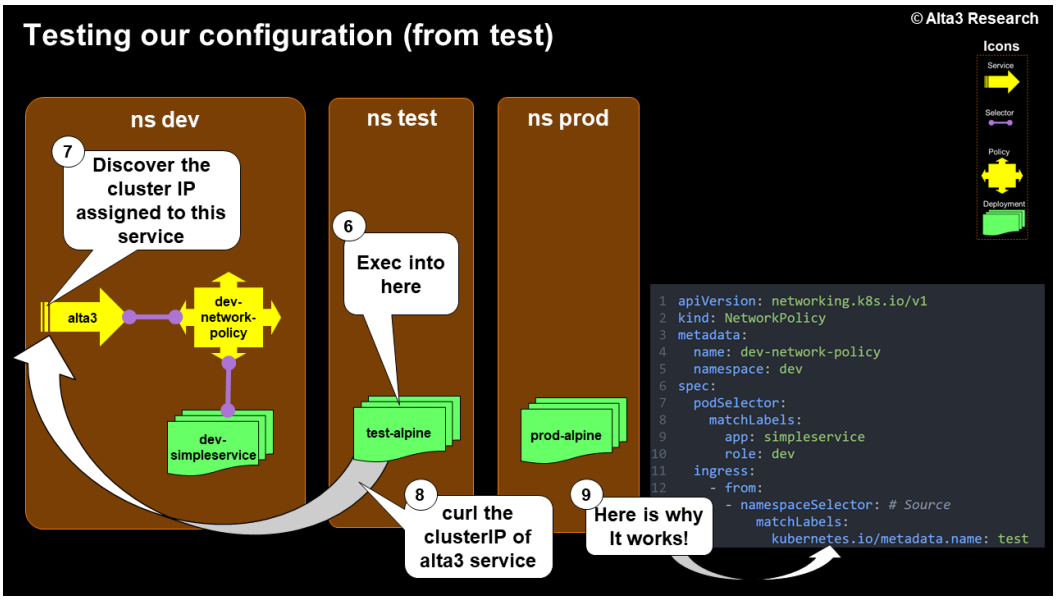
This lab will allow us to control Network Policy on more than just individual pods by enabling us the ability to isolate based on namespace. We'll be using Ingress rules to enact a Network Policy allowing traffic to a pod from a specific namespace. By enacting this policy, we will be denying all other namespaces from ingress to the target pod.

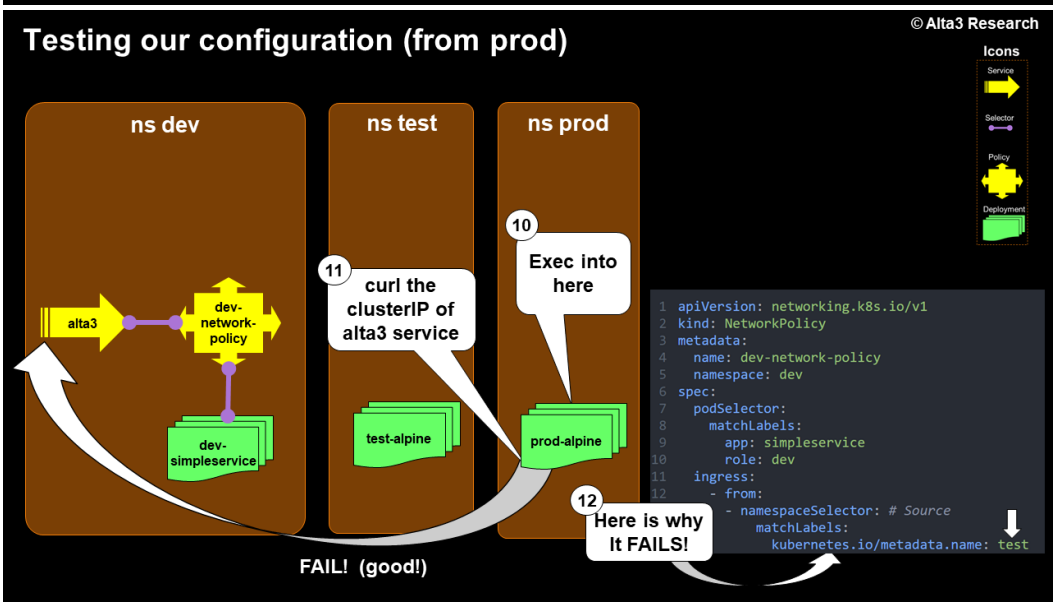
Furthermore, you can also isolate Network Policy targets by specific IP Address Blocks, and Ports.

By the end of this lab, you'll expand your knowledge of Network Policy to include isolation by Namespace.

Procedure - Isolate a Pod from Namespaces







1. To begin, we'll need to build some namespaces! We'll be working with **dev**, **prod**, and the **test** namespaces in this lab.

```
student@bchd:~$
```

```
kubectl create ns dev
kubectl create ns prod
kubectl create ns test
```

2. Create a manifest with a **Deployment** and a **Service** called `dev-deploy.yaml`. Notice this deployment is running a familiar image: Simpleservice. We'll be testing against its endpoints. Furthermore, **pay close attention to the labels used**. We'll be using these labels to select this pod with our **NetworkPolicy**.

```
student@bchd:~$ vim dev-deploy.yaml
```

Click here to view the contents of **dev-deploy.yaml**

Save and quit out of vim with `:wq`

```
student@bchd:~$ kubectl apply -f dev-deploy.yaml

service/alta3 created
deployment.apps/dev-simpleservice created
```

3. Now, we'll create our Network Policy Manifest, which will be using **namespaceSelector** for its ingress policy. This netpolicy allows **ingress** from the **test** namespace. Since **Network Policy** is additive, like **RBAC**, this means only pods from the **test** namespace can connect with it. Be sure to review the manifest closely.

```
student@bchd:~$ vim dev-netpolicy.yaml
```

Click here to view the contents of **dev-netpolicy.yaml**

Save and quit out of vim with `:wq`

```
student@bchd:~$ kubectl apply -f dev-netpolicy.yaml

networkpolicy.networking.k8s.io/dev-network-policy created
```

4. Since we have a deployment in the **dev** namespace, along with a **Network Policy** controlling it, we'll need to test our policy by creating a deployment in both the **test** namespace (named allowed-deploy), and the **prod** namespace (named denied-deploy).

```
student@bchd:~$ vim allowed-deploy.yaml
```

Click here to view the contents of **allowed-deploy.yaml**

Save and quit out of vim with `:wq`

```
student@bchd:~$ kubectl apply -f allowed-deploy.yaml

deployment.apps/test-alpine created
```

5. Now create a deployment in **Production**. Since this deployment is in the **prod** namespace, it will not be able to connect with our target pod in **dev**.

```
student@bchd:~$ vim denied-deploy.yaml
```

Click here to view the contents of **denied-deploy.yaml**

Save and quit out of vim with `:wq`

```
student@bchd:~$ kubectl apply -f denied-deploy.yaml

deployment.apps/prod-alpine created
```

6. With all of our resources in place, it's time we give the **Network Policy** a test! Remember, our Network Policy is only allow ingress to our target pod in **dev** from pods in the **test** namespace. This means our work on **test** should work. Let's begin!

```
student@bchd:~$ kubectl exec -n test -it test-alpine-XXXXXXXXXX-XXXXX -- sh
```

THE ABOVE COMMAND WILL NOT WORK! To obtain the name of the test pod, you have two options: Use tab to complete the pod name, or use `kubectl get pods -n test` to fetch the name of your pod.

7. We are presently on the **test-alpine** pod as root. We'll need to do some work to get it up and running. We'll begin by updating:

```
/ # apk update
```

8. Next, we'll upgrade the pod.

```
/ # apk upgrade
```

9. Finally, we'll install curl.

```
/ # apk add curl
```

10. With all of our setup complete, we're ready to hit one of the endpoints on our **Simpleservice** pod running in the **dev** namespace. But first, we'll need to verify the IP Address. To do this, we should open a new **TMUX** pane.

> Press **CTRL+B** then take your hands off the keyboard. Then press **SHIFT+"** to create a horizontal split.

11. In your new **TMUX Pane** obtain a listing of the services on **dev**.

```
student@bchd:~$ kubectl get svc -n dev

NAME    TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
alta3   ClusterIP   172.16.3.63   <none>        9876/TCP   25m
```

> Note the Cluster-IP Address. This is the IP address of the service which is exposing the simpleservice pod. We'll use this for our curl command. **Your service's ClusterIP address WILL be different**, so use it in the next command.

12. Switch back to your original **TMUX pane**.

> Press **CTRL+B** then take your hands off the keyboard. Then press **UP ARROW"** to switch back to the original pane.

13. Time to let it rip! Remember, this Alpine Pod is on the **test** namespace, so it should be allowed to communicate with our **Simpleservice** pod. **REMEMBER:** replace the IP Address with that of the ClusterIP Address we observed from our **kubectl get svc -n dev** command.

```
/ # curl http://172.16.3.32:9876/env

{"version": "0.5.0", "env": "{'ALTA3_PORT': 'tcp://172.16.3.63:9876', 'ALTA3_PORT_9876_TCP': 'tcp://172.16.3.63:9876',
...
```

> Awesome! Our Alpine pod on the **test** namespace was able to successfully curl against the target pod in the **dev** namespace!

14. We're done with our **test** Alpine pod, so let's hop off of it!

```
/ # exit
```

15. Our next step to ensure the **NetworkPolicy** is working is to test our pod in **prod**. So, let's get this process started!

```
student@bchd:~$ kubectl exec -n prod -it prod-alpine-XXXXXXXXXX-XXXXX -- sh
```

> **THE ABOVE COMMAND WILL NOT WORK!** To obtain the name of the prod pod, you have two options: Use tab to complete the pod name, or use `kubectl get pods -n prod` to fetch the name of your pod.

16. We are presently on the **prod-alpine** pod as root. We'll need to do some work to get it up and running. We'll begin by updating:

```
/ # apk update
```

17. Next, we'll upgrade the pod.

```
/ # apk upgrade
```

18. Finally, we'll install curl.

```
/ # apk add curl
```

19. With all of our setup complete, we're ready to hit one of the endpoints on our **Simpleservice** pod running in the **dev** namespace. But first, we'll need to verify the IP Address. To do this, we should open a new **TMUX** pane.

> Press **CTRL+B** then take your hands off the keyboard. Then press **SHIFT+"** to create a horizontal split.

20. In your new **TMUX Pane** obtain a listing of the services on **dev**.

```
student@bchd:~$ kubectl get svc -n dev

NAME    TYPE        CLUSTER-IP    EXTERNAL-IP   PORT(S)    AGE
alta3   ClusterIP   172.16.3.63   <none>        9876/TCP   25m
```

Note the Cluster-IP Address. This is the IP address of the service which is exposing the simpleservice pod. We'll use this for our curl command. **Your service's ClusterIP address WILL be different**, so use it in the next command.

21. Switch back to your original **TMUX pane**.

Press **CTRL+B** then take your hands off the keyboard. Then press **UP ARROW"** to switch back to the original pane.

22. Time to let it rip! Remember, this Alpine Pod is on the **test** namespace, so it should be allowed to communicate with our **Simpleservice** pod. **REMEMBER:** replace the IP Address with that of the ClusterIP Address we observed from our **kubectl get svc -n dev** command.

```
/ # curl http://172.16.3.32:9876/env
```

Wow! Though not exactly the most exciting result, our attempt to curl has **hung**! This is actually a very, very GOOD thing! This means, we've successfully enabled ingress to our target pod from the **test** namespace, while prohibiting such connections from other namespaces.

23. We're done with our **prod** Alpine pod, so let's hop off of it!

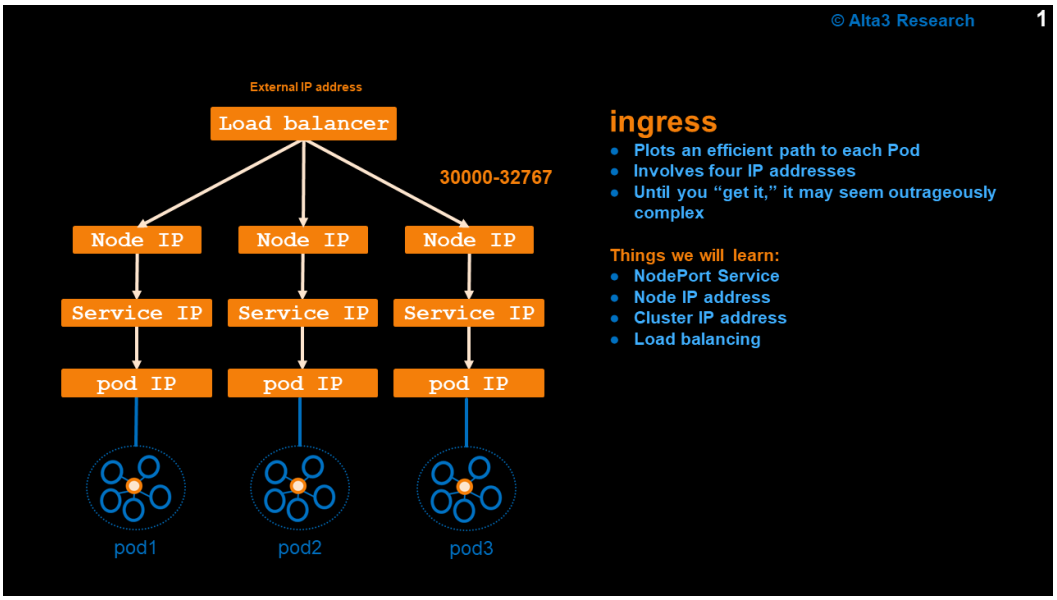
```
/ # exit
```

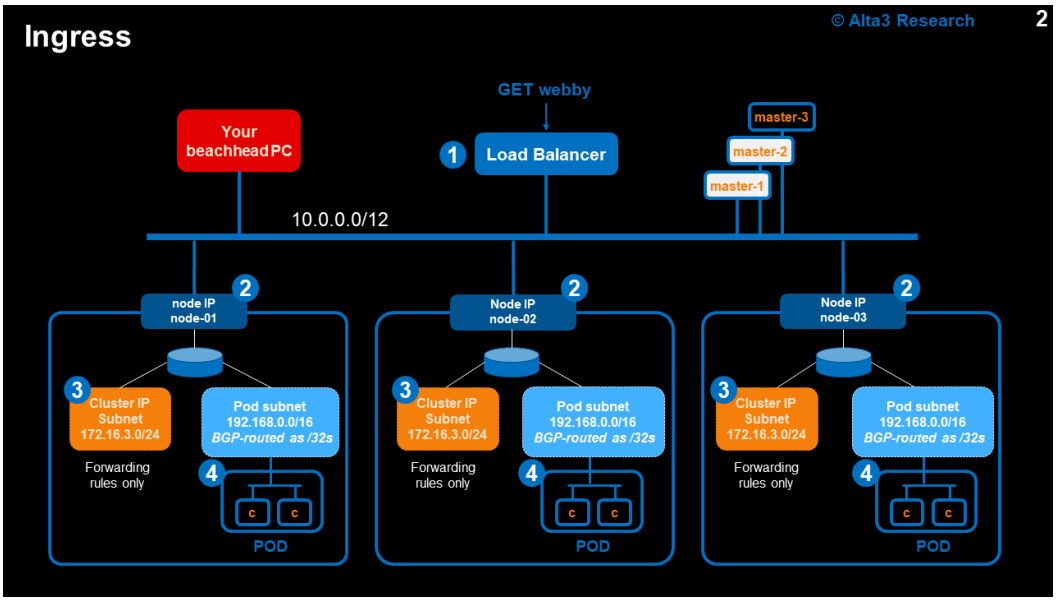
Teardown - Let's tear it all down!

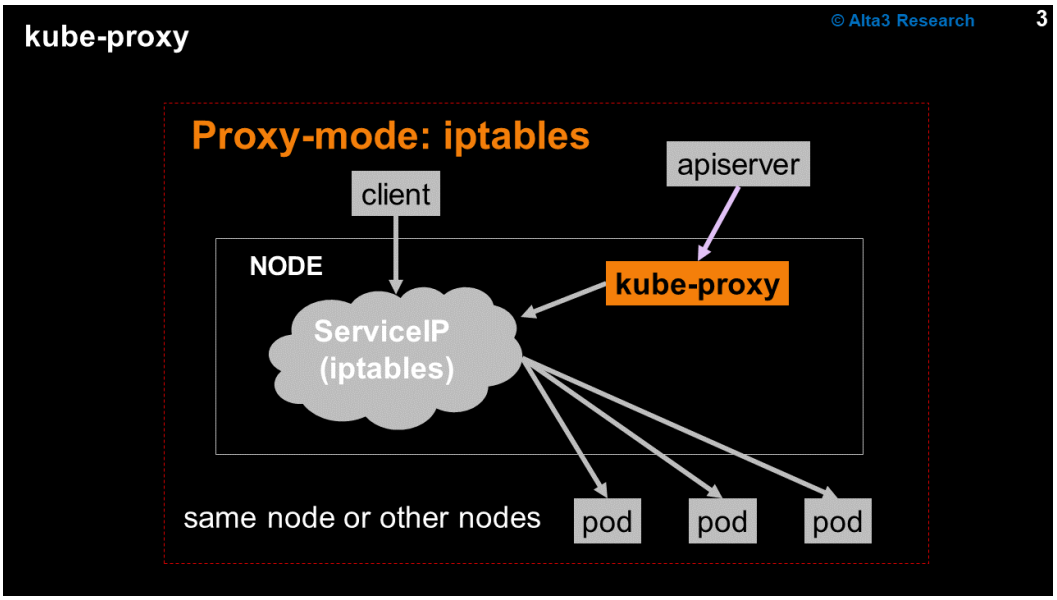
1. Simply run the following commands to tear the lab down, getting ready for the next one.

```
student@bchd:~$

kubectl delete -f dev-deploy.yaml
kubectl delete -f dev-netpolicy.yaml
kubectl delete -f allowed-deploy.yaml
kubectl delete -f denied-deploy.yaml
kubectl delete ns test
kubectl delete ns dev
kubectl delete ns prod
```

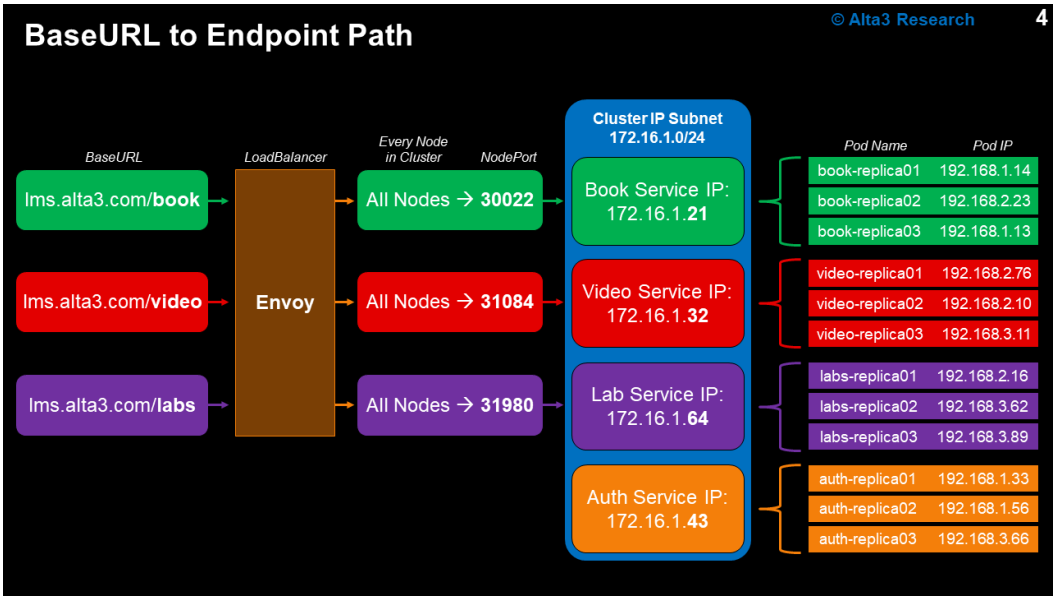
57. Networking

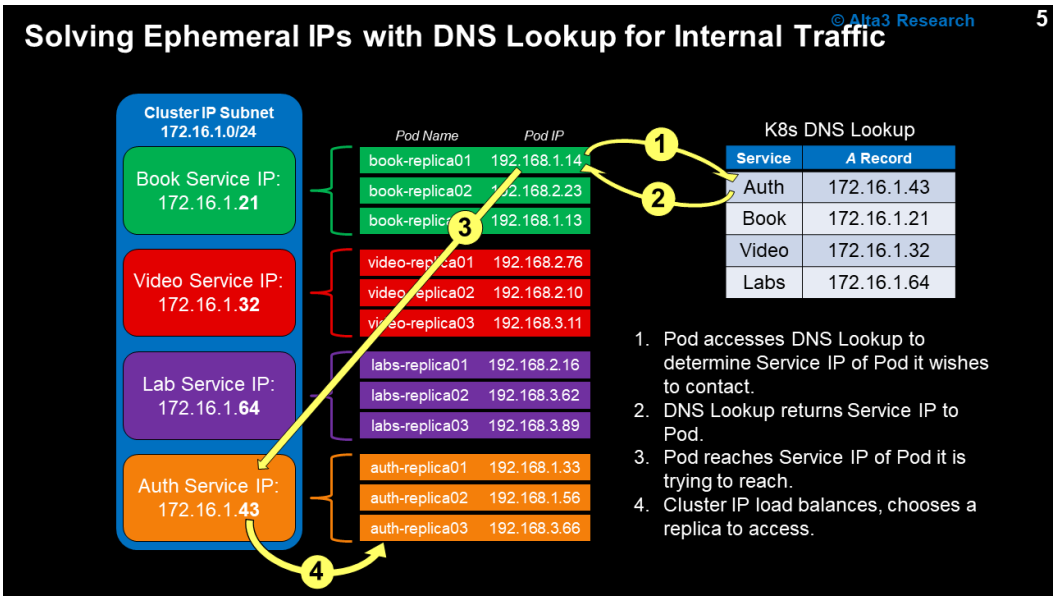






Samarendra Mohapatra
Samarendra.Mohapatra@Viasat.com





Why DNS Should Not Point To Pod IP Addresses

 6

DNS Lookup… if it were to list
EACH INDIVIDUAL POD IP ADDRESS

Service	A Record
Auth	192.168.1.33, 192.168.1.56, 192.168.3.66
Book	192.168.1.14, 192.168.2.23, 192.168.1.13
Video	192.168.2.76, 192.168.2.10, 192.168.3.11
Labs	192.168.2.16, 192.168.3.62, 192.168.3.89

If the DNS lookup worked like this, you'd have to ask:
- Which do I choose?
- How long will a microservice cache the DNS results?
  - *Bad news… normally forever!*
- How do I load balance?
- What happens if the IP address changes?

DNS Lookup
CLUSTER IPs ONLY

Service	A Record
Auth	172.16.1.43
Book	172.16.1.21
Video	172.16.1.32
Labs	172.16.1.64

SOLUTION: Use a Cluster IP Subnet!
- Injects a load balancer
- Don't tell services each others' Pod IPs!

 7

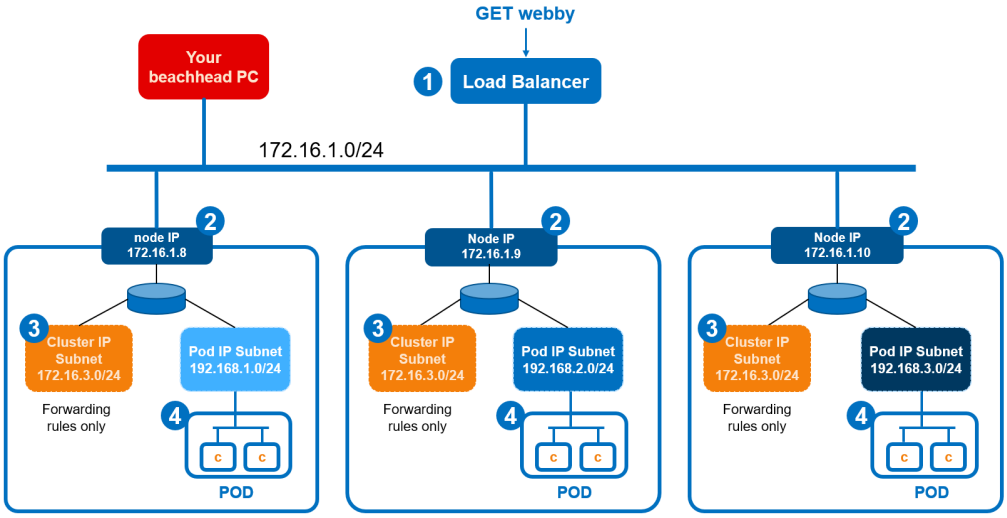
58. Exposing a Service

CKAD Objective

- *Understand Services*

Lab Objective

- Create a service which the service discovery of Kubernetes can look up.
- Allow the application to find and talk with the Kubernetes API.



Just like a Fully Qualified Domain Name (FQDN) is simply a named pointer to a specific IP address at a given time, a Kubernetes Service is a pointer to one or more Pod IP addresses.

All Services terminate at a Pod IP address, otherwise known as an EndPoint. It follows then, that if your service appears 'up', but it is not mapped to an Endpoint IP, chances are the Pod(s) 'down'!

These are the three primary types of Services that you need to be familiar with:

- ClusterIP Service
- NodePort Service
- LoadBalancer Service

1 - ClusterIP Service

A ClusterIP service can only be accessed from inside of the cluster. This means that you, as a Kubernetes Administrator, have the ability to access all of the ClusterIP services via kubectl.

This is accessible via the **spec.clusterIp** port. If a spec.ports[*].targetPort is set it will route from the port to the targetPort. The CLUSTER-IP you get when calling kubectl get services is the IP assigned to this service within the cluster internally.

2 - NodePort Service

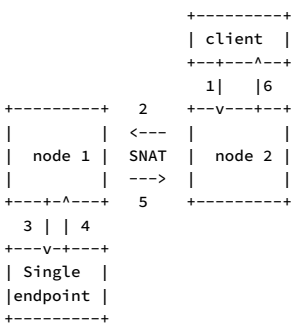
A NodePort Service simply exposes a specific port on **every node in your cluster**, and any traffic that hits that port will then be directed into an EndPoint (Pod) via iptables or netfilter rules.

Your NodeIPs are the external IP addresses of the nodes. If you access this service on a nodePort from the node's external IP, it will route the request to spec.clusterIp:spec.ports[*].port, which will in turn route it to your spec.ports[*].targetPort, if set.

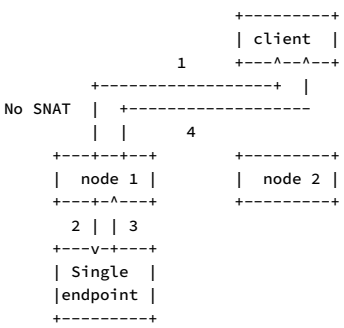
Here are two diagrams to help you see what may happen. If it is not clear to you how a nodePort works, please ask your instructor.

Please Note: These "Single endpoints" represent Pods residing inside of the Node that is pointing to it

Access via node-2 Client to endpoint flow.



Access via node1 Client to endpoint flow



3 - LoadBalancer Service

Most likely, you do not want to be exposing *n* number of Node IP addresses as Public IP addresses. Most cloud providers, such as GKE and AWS, have built load balancers into their cloud that a Kubernetes **LoadBalancer** service is able to dynamically configure.

A **LoadBalancer** service will assure Pods are ready, and then notify the Cloud Load Balancer of the route to the specific Pods. You can access this service from your load balancer's IP address, which routes your request to a nodePort, which in turn routes the request to the ClusterIP (this then load balances to the pods providing that service). You can access this service as you would a NodePort or a ClusterIP service as well.

Envoy (written by Lyft) is typically the load balancer of choice. Unfortunately, Kubernetes has poor support for other External Load Balancers (or maybe other external load balancers have poor support for Kubernetes), so unless you're using Envoy, it is likely you will need to create your own "glue code" to connect the **load Balancer** service to your own Load Balancer.

If you are interested in learning more about Services, check out these other resources:

- **Exposing External IP addresses** - https://kubernetes.io/docs/tutorials/stateless-application/expose-external-ip-address/
- **Basics of Exposing Services** - https://kubernetes.io/docs/tutorials/kubernetes-basics/expose/expose-intro/
- **Understanding Services** - https://kubernetes.io/docs/tasks/access-application-cluster/service-access-application-cluster/

Procedure

1. You made this file in a previous lab, but let's make sure you didn't make any errors in creating it. Run the following command to create file `nginx.conf`.

   ```
   student@bchd:~$ vim ~/nginx.conf
   ```

   Click here to view the contents of **nginx.conf**

2. In a previous lab, we created an nginx.conf config map, but it has changed since then. So, delete the old map.

   ```
   student@bchd:~$ kubectl delete configmap nginx-conf
   ```

3. Now, recreate an updated `nginx-conf` configmap

   ```
   student@bchd:~$ kubectl create configmap nginx-conf --from-file=nginx.conf
   ```

   What is a configMap? We want this new configuration file to be 'injected' into our pod at the time it is created. This behavior takes the place of creating a new image with the proper configuration file in place, as well as manually copying the new file into the pod. First create a **ConfigMap** that includes your configuration data, then describe the config map within the manifest.

4. Once again, you made this file in a previous lab, but let's make sure you didn't make any errors in creating it. Run the following command to create `index.html`. Replace any content with what is shown in the below code block.

```
student@bchd:~$ vim ~/index.html
```

Click here to view the contents of **index.html**

5. Create a ConfigMap for the `index.html` file. This is being created so that the file may be automatically injected into a pod at the time of creation.

```
student@bchd:~$ kubectl create configmap index-file --from-file=index.html
```

6. You may have made this in a previous lab, but in case you missed it, our pod will have a container running an NGINX application, and will expose files it finds within the directory `/var/www/static/` (within the container). Create a text file with clearly identifiable content, the opening lines of George Orwell's *1984* should work.

```
student@bchd:~$ echo "It was a bright cold day in April, and the clocks were striking thirteen." >> nginx.txt
```

7. Create a ConfigMap of our text file.

```
student@bchd:~$ kubectl create configmap nginx-txt --from-file=nginx.txt

  configmap/nginx-txt created
```

8. Verify **all 3** ConfigMaps are loaded.

```
student@bchd:~$ kubectl get cm

NAME         DATA      AGE
index-file   1         1m
nginx-conf   1         2m
nginx-txt    1         5s
```

9. Check to see what pods are currently running.

```
student@bchd:~$ kubectl get pods

NAME                    READY     STATUS    RESTARTS   AGE
nginx-locked-n-loaded   1/1       Running   0          30m
```

10. Clean up your environment by deleting the running pods.

```
student@bchd:~$ kubectl delete pod <POD-NAMES>
```

11. Create your `nginx-configured-expose.yaml` to assure that it looks like the following.

```
student@bchd:~$ vim ~/nginx-configured-expose.yaml
```

Click here to view the contents of **nginx-configured-expose.yaml**

12. Save and exit with `:wq`

13. Create the pod.

```
student@bchd:~$ kubectl apply -f ~/nginx-configured-expose.yaml

pod/nginx-configured
```

14. Confirm which services are currently running. **YOU MAY NOT HAVE ANY/ALL OF THESE SERVICES.**

```
student@bchd:~$ kubectl get services

NAME                     TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)          AGE
docker-private-registry  NodePort    172.16.3.11    <none>        5000:30500/TCP   23h
Kubernetes               ClusterIP   172.16.3.1     <none>        443/TCP          28h
mydb                     ClusterIP   172.16.3.196   <none>        80/TCP           30m
myservice                ClusterIP   172.16.3.189   <none>        80/TCP           30m
simpleservice            ClusterIP   172.16.3.168   <none>        80/TCP           16h
```

15. Expose the pod.

```
student@bchd:~$ kubectl expose pod/nginx-configured --type="NodePort" --port 80

service "nginx-configured" exposed
```

16. Confirm that the new service has been added to the list of services.

```
student@bchd:~$ kubectl get services | grep nginx

nginx-configured         NodePort    172.16.3.67    <none>        80:32034/TCP     76s
```

17. Run the describe services command to see what the output looks like.

```
student@bchd:~$ kubectl describe services/nginx-configured

Name:                     nginx-configured
Namespace:                default
Labels:                   app=nginx-configured
Annotations:              <none>
Selector:                 app=nginx-configured
Type:                     NodePort
IP:                       172.16.3.203
Port:                     <unset>  80/TCP
TargetPort:               80/TCP
NodePort:                 <unset>  32118/TCP
Endpoints:                192.168.2.64:80
Session Affinity:         None
External Traffic Policy:  Cluster
Events:                   <none>
```

18. Get your test pod's node port and store it in an environmental variable called `NODE_PORT`.

```
student@bchd:~$ export NODE_PORT=$(kubectl get services/nginx-configured -o go-template='{{(index .spec.ports 0).nodePort}}')
```

19. For informational purposes, determine what NODE `nginx-configured` is running on.

```
student@bchd:~$ kubectl describe pod nginx-configured | grep "Node:\|IP:"
```

20. Check out your nodeport's port.

```
student@bchd:~$ echo Your NODE_PORT = $NODE_PORT

Your NODE_PORT=30112
```

21. Answer the following questions:

    1. **Question: What node is your pod actually running on?**
        - *Answer:* `kubectl get pods -o wide` *or* `kubectl describe pods` *can help you find those.*
    2. **Question: If you curl the service's NodePort on the nodes that are not hosting the pod, will you STILL get a response?**
        - *Answer: Yes- the last few lab directions show you what happens.*
    3. **Question: What is the IP of each node? (node-1, node-2)?**
        - *Answer:* `kubectl describe nodes | grep IP` *should do the trick!*
    4. **Question: If you ping node-1 and node-2 does it resolve to the expected IP address?**
        - *Answer: It should resolve correctly.*
    5. **Question: What TCP socket would you curl to test if you can access your service at node-1?**
        - *Answer: The one that is listed as the NodePort.*
    6. **Question: What TCP socket would you curl to test if you can access your service at node-2?**
        - *Answer: Same as answer 5.*

22. Do **NOT** proceed without answering all of the questions above.

23. The following command will curl node-1 on your service's nodeport.

```
student@bchd:~$ curl node-1:$NODE_PORT/static/nginx.txt
```

> If the above command fails, see if your pod is running with "kubectl get pods". If the pod is failing to come up, delete it with "kubectl delete -f nginx-configured.yaml". Next, check to ensure your ConfigMaps are in place with "kubectl get cm". If you do not have all 3 configmaps, or if they were made in correctly, your Pod will fail to boot. To fix, try deleting all of your ConfigMaps, recreating them, and then recreating your pod.

24. The following command will curl node-2 on your service's nodeport.

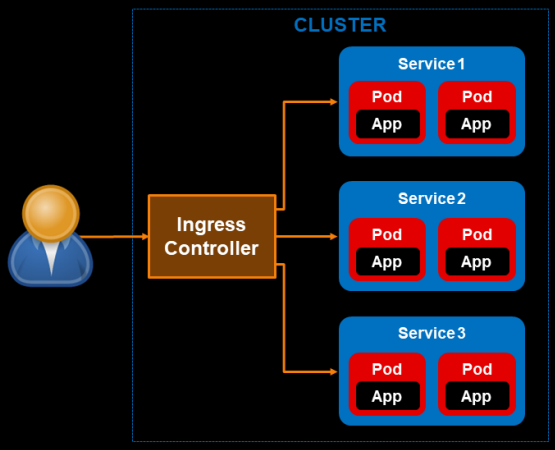
```
student@bchd:~$ curl node-2:$NODE_PORT/static/nginx.txt
```

59. Ingress Controllers

Ingress Controllers

 1

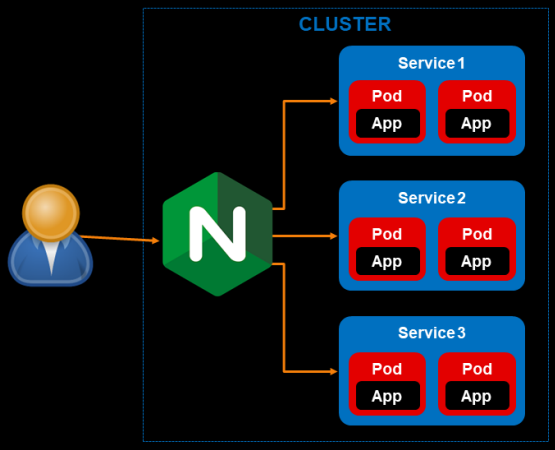
- Ingress exposes HTTP and HTTPS routes from outside the cluster to services within the cluster. An Ingress controller is required for this to work.
- Ingress controllers are third party proxies, NGINX and GCE being two.
- Controllers require an external load balancer to work, and they add another layer of routing and control.



NGINX Ingress Controller

 2

- Uses a 100% Nginx instance, no third-party modules needed.
- Allows annotations to do things like define load balancing algorithms and persistence



Ingress Manifest

 3

(NOTE: You cannot execute Ingress without an Ingress controller)

Controls behavior of Ingress controller

OPTIONAL- provide host that these rules apply to. If not specified, all inbound HTTP traffic is affected.

List of paths with their associated backend. This must match an incoming request before the load balancer directs traffic to the referenced Service.

HTTP (and HTTPS) requests to the Ingress that matches the host and path of the rule are sent to the listed backend.

```
apiVersion: networking.k8s.io/v1beta1
kind: Ingress
metadata:
  name: test-ingress
  annotations:
    nginx.ingress.kubernetes.io/rewrite-target: /
spec:
  rules:
  - host: foo.bar.com
    http:
      paths:
      - path: /foo
        backend:
          serviceName: service1
          servicePort: 4200
      - path: /bar
        backend:
          serviceName: service2
          servicePort: 8080
```

Understanding Layer 7 Routing

 4

Vocabulary words:

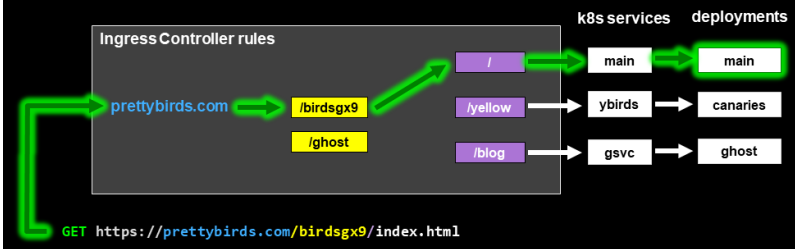
method - GET, POST, PUT, PATCH, DELETE
scheme - http:// or https://
FQDN - Resolves to an IP address (hostname + domain name)
base path - Points to specific microservice(s)
context path - Supporting message routing INSIDE a microservice
file - The file requested by the GET

```
GET https://prettybirds.com/birdsgx9/index.html
GET https://prettybirds.com/birdsgx9/yellow/canaries.html
GET https://prettybirds.com/birdsgx9/blue/bluebird.html
```

method	scheme	fqdn	base path	context path	file
GET	https://	labs.alta3.com	/birdsgx9	/	index.html
GET	https://	labs.alta3.com	/birdsgx9	/yellow/	canaries.html
GET	https://	labs.alta3.com	/birdsgx9	/blue/	bluebird.html

Layer 7 Routing step by step (1 of 3)

 5



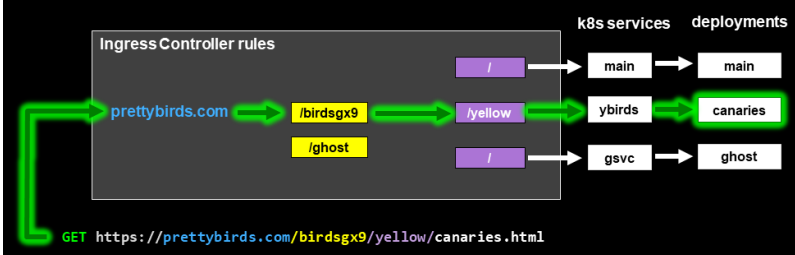
method	scheme	host	base path	context path	file
GET	https://	labs.alta3.com	/birdsgx9	/	index.html
GET	https://	labs.alta3.com	/birdsgx9	/yellow/	canaries.html
GET	https://	labs.alta3.com	/ghost	/	index.html

Ingress manifest

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "prettybirds.com"
    http:
      paths:
      - pathType: Prefix
        path: "/birdsgx9/"
        backend:
          service:
            name: main
            port:
              number: 80
      - pathType: Prefix
        path: "/birdsgx9/yellow/"
        backend:
          service:
            name: ybirds
            port:
              number: 80
      - pathType: Prefix
        path: "/ghost/"
        backend:
          service:
            name: gsvc
            port:
              number: 80
```

Layer 7 Routing step by step (2 of 3)

© Alta3 Research 6




```
GET https://prettybirds.com/birdsgx9/yellow/canaries.html
```

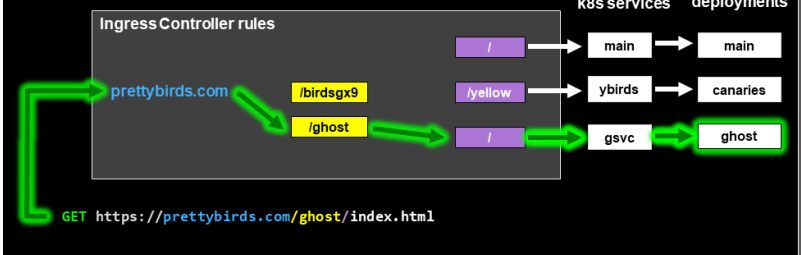
method	scheme	host	base path	context path	file
GET	https://	labs.alta3.com	/birdsgx9	/	index.html
GET	https://	labs.alta3.com	/birdsgx9	/yellow/	canaries.html
GET	https://	labs.alta3.com	/ghost	/	index.html

Ingress manifest

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "prettybirds.com"
    http:
      paths:
      - pathType: Prefix
        path: "/birdsgx9/"
        backend:
          service:
            name: main
            port:
              number: 80
      - pathType: Prefix
        path: "/birdsgx9/yellow/"
        backend:
          service:
            name: ybirds
            port:
              number: 80
      - pathType: Prefix
        path: "/ghost/"
        backend:
          service:
            name: gsvc
            port:
              number: 80
```

Layer 7 Routing step by step (3 of 3)

© Alta3 Research 7




```
GET https://prettybirds.com/ghost/index.html
```

method	scheme	host	base path	context path	file
GET	https://	labs.alta3.com	/birdsgx9	/	index.html
GET	https://	labs.alta3.com	/birdsgx9	/yellow/	canaries.html
GET	https://	labs.alta3.com	/ghost	/	index.html

Note that ghost (blogging service) needs an entire " / " context path! This is no problem if a **unique base path** is configured to distinguish the birds microservices from the blogging service context!

Ingress manifest

```
apiVersion: networking.k8s.io/v1
kind: Ingress
metadata:
  name: ingress-wildcard-host
spec:
  rules:
  - host: "prettybirds.com"
    http:
      paths:
      - pathType: Prefix
        path: "/birdsgx9/"
        backend:
          service:
            name: main
            port:
              number: 80
      - pathType: Prefix
        path: "/birdsgx9/yellow/"
        backend:
          service:
            name: ybirds
            port:
              number: 80
      - pathType: Prefix
        path: "/ghost/"
        backend:
          service:
            name: gsvc
            port:
              number: 80
```

60. Expose a Service Via Ingress

Lab Objective

Create Services containing the endpoints of Pods. Create Ingress resource to expose Services.

In Kubernetes, an Ingress is a special type of service that allows access to your Kubernetes services from outside the Kubernetes cluster. You configure access by creating a collection of rules that define which inbound connections reach which other services within the cluster.

This lets you consolidate your routing rules into a single resource. For example, you might want to send requests to example.com/api/v1/ to an api-v1 service, and requests to example.com/api/v2/ to the api-v2 service. With an Ingress, you can easily set this up without creating multiple LoadBalancers or NodePorts. Both LoadBalancers and NodePorts can represent finite resources and costs depending on the hosting and network providers of your Kubernetes cluster.

NodePort and LoadBalancer let you expose a service by specifying that value in the service's type. Ingress, on the other hand, is a completely independent resource to your service. You can declare, create, modify, and destroy it separately to your services. This makes it decoupled and isolated from the services you want to expose. It also helps you to consolidate routing rules into one place.

One downside is that you will need to configure an Ingress Controller for your cluster. But that's pretty easy— in this example, we'll use the Nginx Ingress Controller.

Procedure

1. Installing the Nginx Ingress Controller may vary depending on the provider that you are using (AWS, GCE-GKE, Azure... read about them here: https://kubernetes.github.io/ingress-nginx/deploy/.

```
student@bchd:~$ kubectl apply -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.2/deploy/static/provider/baremetal/deploy.yaml

namespace/ingress-nginx created
serviceaccount/ingress-nginx created
serviceaccount/ingress-nginx-admission created
role.rbac.authorization.k8s.io/ingress-nginx created
role.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrole.rbac.authorization.k8s.io/ingress-nginx created
clusterrole.rbac.authorization.k8s.io/ingress-nginx-admission created
rolebinding.rbac.authorization.k8s.io/ingress-nginx created
rolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx created
clusterrolebinding.rbac.authorization.k8s.io/ingress-nginx-admission created
configmap/ingress-nginx-controller created
service/ingress-nginx-controller created
service/ingress-nginx-controller-admission created
deployment.apps/ingress-nginx-controller created
job.batch/ingress-nginx-admission-create created
job.batch/ingress-nginx-admission-patch created
ingressclass.networking.k8s.io/nginx created
validatingwebhookconfiguration.admissionregistration.k8s.io/ingress-nginx-admission created
```

2. Let's confirm the new services running on our cluster.

```
student@bchd:~$ kubectl get services --all-namespaces | grep ingress-nginx

ingress-nginx-controller             NodePort    172.16.3.181   <none>        80:31566/TCP,443:32571/TCP   149m
ingress-nginx-controller-admission   ClusterIP   172.16.3.141   <none>        443/TCP                      149m
```

3. Let's confirm that the ingress controller pods are running:

```
student@bchd:~$ kubectl get pods --namespace=ingress-nginx

NAME                                        READY   STATUS      RESTARTS   AGE
ingress-nginx-admission-create-6kv5w        0/1     Completed   0          148m
ingress-nginx-admission-patch-nbwjj         0/1     Completed   1          148m
ingress-nginx-controller-5bf7467b67-gltpg   1/1     Running     0          148m
```

4. Let's use a quick and dirty command to spin up a deployment named `demo` using the `httpd` image. It will only create one replica.

```
student@bchd:~$ kubectl create deployment demo --image=httpd --port=80
```

5. The `demo` deployment should now have a pod running. Expose all the `demo` deployment's pods with a `kubectl expose` command. This will create a ClusterIP service for our deployment.

```
student@bchd:~$ kubectl expose deployment demo
```

6. This next part will be challenging. Open your Group Dashboard page in Alta3 Live in a new tab.

7. Click on the `Assigned to You` dropdown, then select `aux1`. Open `aux1` in a new tab.

8. Copy the URL of the `aux1` page out of your browser. It should look something like this:

```
https://aux1-RANDOM-STRING-RANDOM-STRING-JUSTANEXAMPLE.live.alta3.com/
```

9. DON'T use the `https://` at the beginning of the URL. Use it to complete the command below and run the command.

```
student@bchd:~$ kubectl create ingress demo-localhost --class=nginx --rule="aux1-YOURS-GOES-HERE.live.alta3.com/*=demo:80"
```

10. Once you've completed that step without error, check out the ingress object you just created.

```
student@bchd:~$ kubectl describe ingress demo-localhost

Name:             demo-localhost
Labels:           <none>
Namespace:        default
Address:          10.4.144.33
Default backend:  default-http-backend:80
Rules:
  Host                                                  Path  Backends
  ----                                                  ----  --------
  aux1-fb22ff6b-ac75-4da3-8457-efcb02d3bb97.live.alta3.com
                                                        /   demo:80 (192.168.84.151:80)
Annotations:                                            <none>
Events:                                                 <none>
```

11. The next thing we need to do is configure one of our nodes to act as the point of external contact. This could be done behind a LoadBalancer, or as a specific public IP address. We need to get the nodeport associated with port 443. This is shown to the right of `443:` after issuing the command below.

```
student@bchd:~$ kubectl get service ingress-nginx-controller -n ingress-nginx

NAME                       TYPE       CLUSTER-IP     EXTERNAL-IP   PORT(S)                      AGE
ingress-nginx-controller   NodePort   172.16.3.181   <none>        80:31566/TCP,443:32571/TCP   3h2m
```

> In this example, the nodeport we're looking for is 32571, NOT 31566!

12. Set up a reverse proxy on your beachhead that will service inbound requests to the `aux1` student path. Notice we are load balancing the request across all nodes in the cluster and targeting the ingress service. Ensure you correctly replace and assign `<NODE_PORT>` for the https (443) node port assigned to the ingress service.

```
student@bchd:~$ sudo vim /etc/nginx/sites-enabled/demo-ingress

upstream nodes {
  server node-1:NODE_PORT;
  server node-2:NODE_PORT;
}

server {

  listen 2224 default_server;
  listen [::]:2224 default_server;

  location / {
    proxy_pass https://nodes;
    proxy_set_header Host $http_host;
  }
}
```

13. Finally, we need to test and reload the local host's nginx configuration.

```
student@bchd:~$ sudo nginx -t

student@bchd:~$ sudo nginx -s reload
```

14. Return to `Group Dashboard` and under `Assigned to You` select `aux1` from the dropdown. Your `aux1` is now the path to your application and should display "It works!" If you do not see this message try clearing your browser's cache data.

15. **MANDATORY CLEANUP**- Before ending this lab, you'll want to clear up the changes we made to our local nginx instance. We won't be using this ingress in future labs and it will prevent you from using port 2224 for port-forwarding!

```
student@bchd:~$ sudo mv /etc/nginx/sites-enabled/demo-ingress /etc/nginx/sites-available/demo-ingress

student@bchd:~$ sudo nginx -t

student@bchd:~$ sudo nginx -s reload
```

16. You'll likely want to clean up the following as well, since we won't use them in future labs.

```
student@bchd:~$ kubectl delete deployments.apps demo

student@bchd:~$ kubectl delete ingress demo-localhost

student@bchd:~$ kubectl delete namespace ingress-nginx
```

Optional Cleanup to remove Ingress Controller

```
student@bchd:~$ kubectl delete -f https://raw.githubusercontent.com/kubernetes/ingress-nginx/controller-v1.1.2/deploy/static/provider/cloud/deploy.yaml
```

```
---
apiVersion: v1
kind: Pod
metadata:
  name: webby-nginx-combo
  labels:
    app: nginx-configured
spec:
  containers:
  - name: webby
    image: registry.gitlab.com/alta3/webby:latest
    ports:
    - containerPort: 8888
  - name: nginx
    image: nginx:1.18.0
    ports:
      - containerPort: 80
      - containerPort: 443
    volumeMounts:
    - name: nginx-proxy-config
      mountPath: /etc/nginx/nginx.conf
      subPath: nginx.conf
    - name: my-index-file
      mountPath: /var/www/index.html
      subPath: index.html
    - name: static-demo-data
      mountPath: /var/www/static/nginx.txt
      subPath: nginx.txt
  volumes:
  - name: nginx-proxy-config
    configMap:
      name: nginx-conf
  - name: my-index-file
    configMap:
      name: index-file
  - name: static-demo-data
    configMap:
      name: nginx-txt
```

1. Confirm you have the correct ConfigMaps by running the following command:

```
student@bchd:~$ bash ~/git/kubernetes-the-alta3-way/labs/scripts/cmsetup.sh
```

Procedure

1. Create a new chart called `webby-chart` with Helm.

```
student@bchd:~$ helm create webby-chart

Creating webby-chart
```

2. Remove all of the current templates.

```
student@bchd:~$ rm -rf ~/webby-chart/templates/*
```

3. Copy your composite deployment manifest into the templates directory.

```
student@bchd:~$ cp webby-nginx-combo.yaml ~/webby-chart/templates/
```

4. Now we have to get all of our ConfigMaps exported into the templates directory as well. First ensure you have the **index-file**

```
student@bchd:~$ kubectl get cm index-file -o yaml > ~/webby-chart/templates/index-file.yaml
```

5. Great! Now ensure you have the **nginx-conf**

```
student@bchd:~$ kubectl get cm nginx-conf -o yaml > ~/webby-chart/templates/nginx-conf.yaml
```

6. Finally, dump a copy of the configMap **nginx-txt**

```
student@bchd:~$ kubectl get cm nginx-txt -o yaml > ~/webby-chart/templates/nginx-txt.yaml
```

7. Now we need to make sure that we have this chart working properly. To do so, let's clean out our cluster and start over again with Helm. First, delete all of the ConfigMaps.

```
student@bchd:~$ kubectl delete cm --all
```

8. Next, delete a deployment.

```
student@bchd:~$ kubectl delete deploy nginx-webby
```

9. Verify that there are no Pods, Deployments, or ConfigMaps running. If so, delete them. (You may likely have random pods from previous labs running still).

```
student@bchd:~$ kubectl get pods,deploy,cm
```

10. Now is the moment you have been waiting for. It is time to deploy your Chart with Helm and verify that it is working.

```
student@bchd:~$ helm install ./webby-chart --generate-name

NAME: webby-chart-1587003842
LAST DEPLOYED: Wed Apr 15 22:24:03 2020
NAMESPACE: default
STATUS: deployed
REVISION: 1
TEST SUITE: None
```

11. Let's check out the pods that are now running from the helm deployment. Pretty cool, huh?

```
student@bchd:~$ kubectl get pods
```

12. Verify that you can access the website by port-forwarding one of the pods and curling your localhost.

```
student@bchd:~$ kubectl port-forward webby-nginx-combo 2224:80 --address=0.0.0.0
```

curl

```
student@bchd:~$ curl localhost:2224
```

13. Awesome! Great job! Now package up your chart so that you have it recorded somewhere nice and neat.

```
student@bchd:~$ helm package webby-chart

Successfully packaged chart and saved it to: /home/student/webby-chart-0.1.0.tgz
```

> Helm will default to adding this new chart to a local repository. The way to install this chart again would be to run `helm install local/webby-chart-0.1.0.tgz`

61. Patching

Lab Objective

This lab is divided into two parts. First you will learn the different methods of patching Kubernetes API Objects without the need to redeploy or kill off existing objects that are currently running. The second part of this lab will focus on some of the other aspects of patching, such as patching a node for maintenance.

Procedure

Strategic Merge Patch

1. Let's create a deployment manifest to launch.

```
student@bchd:~$ vim ~/deployment-patch.yaml
```

Click here to view the contents of **deployment-patch.yaml**

2. Now use kubectl to create the deployment.

```
student@bchd:~$ kubectl apply -f ~/deployment-patch.yaml

deployment.apps/patch-demo created
```

3. View the pods that have been associated with your new deployment. This step shows a simple method to view all pods, but try finding a way to view only the pods on this particular deployment.

```
student@bchd:~$ kubectl get pods

NAME                          READY   STATUS    RESTARTS   AGE
patch-demo-6d6b8fd787-qxqnv   1/1     Running   0          97s
patch-demo-6d6b8fd787-sknfz   1/1     Running   0          97s
```

4. Now create a new file named 'patch-file.yaml'.

```
student@bchd:~$ vim ~/patch-file.yaml
```

Click here to view the contents of **patch-file.yaml**

5. Let's patch our deployment.

```
student@bchd:~$ kubectl patch deployment patch-demo --patch "$(cat patch-file.yaml)"

deployments.apps/patch-demo patched
```

6. View the patched deployment. Do you see the containers and the multiple images now?

```
student@bchd:~$ kubectl get deployment patch-demo --output wide

NAME         READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS                       IMAGES        SELECTOR
patch-demo   2/2     2            2           9m45s   patch-demo-ctr-2,patch-demo-ctr   redis,nginx   app=nginx
```

7. Take a look at the yaml output of the newly patched deployment. We want to highlight the containers that we saw above with some extra details. If you want to see the entire output, it is below.

```
student@bchd:~$ kubectl get deployment patch-demo -o yaml
```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "2"
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"name":"patch-demo","namespace":"default"},"spec":{"replicas":
2,"selector":{"matchLabels":{"app":"nginx"}},"template":{"metadata":{"labels":{"app":"nginx"}},"spec":{"containers":[{"image":"nginx:
1.19.6","name":"patch-demo-ctr"}],"tolerations":[{"effect":"NoSchedule","key":"dedicated","value":"test-team"}]}}}}
  creationTimestamp: "2022-06-27T20:31:46Z"
  generation: 2
  name: patch-demo
  namespace: default
  resourceVersion: "22033"
  selfLink: /apis/apps/v1/namespaces/default/deployments/patch-demo
  uid: 6969e496-de0f-4bef-bb78-0468987c67f0
spec:
  progressDeadlineSeconds: 600
  replicas: 2
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:
      - image: redis:7.0.2
        imagePullPolicy: IfNotPresent
        name: patch-demo-ctr-2
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      - image: nginx:1.19.6
        imagePullPolicy: IfNotPresent
        name: patch-demo-ctr
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
      tolerations:
      - effect: NoSchedule
        key: dedicated
        value: test-team
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: "2022-06-27T20:31:48Z"
    lastUpdateTime: "2022-06-27T20:31:48Z"
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: "2022-06-27T20:31:46Z"
    lastUpdateTime: "2022-06-27T20:32:18Z"
```

```
    message: ReplicaSet "patch-demo-84fb9557d7" has successfully progressed.
    reason: NewReplicaSetAvailable
    status: "True"
    type: Progressing
  observedGeneration: 2
  readyReplicas: 2
  replicas: 2
  updatedReplicas: 2
```

8. Take a look at your pods again.

```
student@bchd:~$ kubectl get pods

NAME                          READY   STATUS    RESTARTS   AGE
patch-demo-c8796b6dd-cfn58   2/2     Running   0          30m
patch-demo-c8796b6dd-l47dc   2/2     Running   0          30m
```

9. Get a closer look into one of the pods above. Remember to replace "<pod-name>" with the actual name of your pod.

```
student@bchd:~$ kubectl get pod <pod-name> -o yaml
```

10. The event log is available to us as well, which is always worth checking out when you have an issue. Again, the output is going to be large. We are specifying a single pod that we want to check the output for x number of reasons. You can also do a describe here and look toward the bottom, where the events section is located.

```
student@bchd:~$ kubectl describe pod <pod-name>
```

11. Unsure what you are looking for? Here is a snippet (example) of the event logs. This should appear near the bottom of the output generated by the `kubectl describe pod` command.

```
Events:
  Type    Reason     Age   From               Message
  ----    ------     ----  ----               -------
  Normal  Scheduled  3m8s  default-scheduler  Successfully assigned default/patch-demo-84fb9557d7-jvsk2 to node-1
  Normal  Pulling    3m7s  kubelet            Pulling image "redis:7.0.2"
  Normal  Pulled     3m6s  kubelet            Successfully pulled image "redis:7.0.2" in 1.483948185s
  Normal  Created    3m6s  kubelet            Created container patch-demo-ctr-2
  Normal  Started    3m6s  kubelet            Started container patch-demo-ctr-2
  Normal  Pulled     3m6s  kubelet            Container image "nginx:1.19.6" already present on machine
  Normal  Created    3m6s  kubelet            Created container patch-demo-ctr
  Normal  Started    3m6s  kubelet            Started container patch-demo-ctr
```

12. Make another patch for this deployment. This time we're going to use tolerations. Create a file named patch-file-tolerations.yaml.

```
student@bchd:~$ vim ~/patch-file-tolerations.yaml
```

Click here to view the contents of **patch-file-tolerations.yaml**

13. Patch your deployment.

```
student@bchd:~$ kubectl patch deployment patch-demo --patch "$(cat patch-file-tolerations.yaml)"

deployment.apps/patch-demo patched
```

14. View the patched Deployment and try to find the output shown below. It should be located further down the list. Just keep hitting `Enter` until you see it. Or as a quicker method use a `/` and type in tolerations, then hit `Enter` to highlight that section. Hit `q` to quit out of less.

```
student@bchd:~$ kubectl get deployment patch-demo -o yaml | less

tolerations:
      - effect: NoSchedule
        key: disktype
        value: ssd
```

15. Congratulations! You've just completed what is called a strategic merge patch. But we're not done yet. Next, we are going to dive into a JSON merge patch. One key to note is the value of the patchStrategy key within the k8s source code. Ah, you don't see it do you? It is not required but because of this, the strategic merge patch uses the default patch strategy of *replace*.

JSON Merge Patch

16. So we just learned about the strategic merge patch. Guess what? The JSON merge patch is just a slight deviation from what we just learned. We will use the same deployment for this part of the lab. If you want a clean slate, just delete the deployment and make sure it is actually removed. Now deploy it again.

    One item mentioned was regarding the patchStrategy. The kubectl patch command has a type parameter that can be set to one of three values. As you finish this lab, try to take note of which parameter and related merge type goes with that. Here is a chart to preview.

    ```
    | Parameter Value | Merge Type                |
    | ---             | ---                       |
    | json            | JSON Patch, RFC 6902      |
    | merge           | JSON Merge Patch, RFC7386 |
    | strategic       | Strategic merge patch     |
    ```

17. First, we need to create a new file called patch-file-2.yaml

    ```
    student@bchd:~$ vim ~/patch-file-2.yaml
    ```

    Click here to view the contents of **patch-file-2.yaml**

18. Setting the type to *merge*, patch the deployment.

    ```
    student@bchd:~$ kubectl patch deployment patch-demo --type merge --patch "$(cat patch-file-2.yaml)"

    deployment.apps/patch-demo patched
    ```

19. Using the skills you learned above, view the patched deployment.

    ```
    student@bchd:~$ kubectl get deployment patch-demo --output wide

    NAME         READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS         IMAGES                                SELECTOR
    patch-demo   2/2     2            2           79m   patch-demo-ctr-3   gcr.io/google-samples/node-hello:1.0   app=nginx

    student@bchd:~$ kubectl get deployment patch-demo -o yaml
    ```

```
apiVersion: apps/v1
kind: Deployment
metadata:
  annotations:
    deployment.kubernetes.io/revision: "4"
    kubectl.kubernetes.io/last-applied-configuration: |
      {"apiVersion":"apps/v1","kind":"Deployment","metadata":{"annotations":{},"name":"patch-demo","namespace":"default"},
"spec":{"replicas":2,"selector":{"matchLabels":{"app":"nginx"}},"template":{"metadata":{"labels":{"app":"nginx"}},"spec":{
"containers":[{"image":"nginx","name":"patch-demo-ctr"}],"tolerations":[{"effect":"NoSchedule","key":"dedicated","value":"
test-team"}]}}}}
  creationTimestamp: "2020-03-24T19:09:34Z"
  generation: 4
  name: patch-demo
  namespace: default
  resourceVersion: "133060"
  selfLink: /apis/apps/v1/namespaces/default/deployments/patch-demo
  uid: 0c55d745-9481-422c-a125-1975ba00532f
spec:
  progressDeadlineSeconds: 600
  replicas: 2
  revisionHistoryLimit: 10
  selector:
    matchLabels:
      app: nginx
  strategy:
    rollingUpdate:
      maxSurge: 25%
      maxUnavailable: 25%
    type: RollingUpdate
  template:
    metadata:
      creationTimestamp: null
      labels:
        app: nginx
    spec:
      containers:                                    #<--- containers
      - image: gcr.io/google-samples/node-hello:1.0   #<--- only "-" under with image
        imagePullPolicy: IfNotPresent
        name: patch-demo-ctr-3
        resources: {}
        terminationMessagePath: /dev/termination-log
        terminationMessagePolicy: File
      dnsPolicy: ClusterFirst
      restartPolicy: Always
      schedulerName: default-scheduler
      securityContext: {}
      terminationGracePeriodSeconds: 30
      tolerations:
      - effect: NoSchedule
        key: disktype
        value: ssd
status:
  availableReplicas: 2
  conditions:
  - lastTransitionTime: "2020-03-24T19:10:22Z"
    lastUpdateTime: "2020-03-24T19:10:22Z"
    message: Deployment has minimum availability.
    reason: MinimumReplicasAvailable
    status: "True"
    type: Available
  - lastTransitionTime: "2020-03-24T19:09:36Z"
    lastUpdateTime: "2020-03-24T20:29:35Z"
    message: ReplicaSet "patch-demo-795c95f476" has successfully progressed.
    reason: NewReplicaSetAvailable
    status: "True"
    type: Progressing
  observedGeneration: 4
```

```
readyReplicas: 2
replicas: 2
updatedReplicas: 2
```

As you can see above from either of the options, the new pods are only running one container each.

Alternative Patching

20. You've just learned about JSON and strategic patching. Below, we'll take a quick look at alternate forms of the kubectl patch command. Create a new file called patch-file.json.

```
student@bchd:~$ vim ~/patch-file.json
```

Click here to view the contents of **patch-file.json**

21. You can use the following commands (which are equivalent) to patch in our change. Which do you like better?

```
student@bchd:~$ kubectl patch deployment patch-demo --patch "$(cat patch-file.json)"
```

```
student@bchd:~$ kubectl patch deployment patch-demo --patch '{"spec": {"template": {"spec": {"containers": [{"name": "patch-demo-ctr-2","image": "redis:7.0.2"}]}}}}'
```

22. All done for this lab. In this exercise we used the *kubectl patch* command to change the live configuration of a deployment object. You did not have to change the original deployment configuration file to do so.

62. Inspect Container Logging

CKAD Objective

- Understand container logging

When your application needs debugging, it's helpful to be able to dig deeper into what the application is doing. The ability to access the logs of a pod in your Kubernetes cluster is quite easy. Kubernetes provides the `logs` command for debugging and troubleshooting running containers.

Lab Objective

Review the logs of a pod and see what this command actually offers. It may be more than you need, or it may be less than you need.

Access the logs of a pod that is currently running to see the information available to you.

Procedure

1. First, we need to create a new pod in order to check it's logs. Create your basic nginx pod again. If you misplaced the filename, here is the Pod Manifest you should be using.

```
apiVersion: v1
kind: Pod
metadata:
  name: nginx
spec:
  containers:
  - name: nginx
    image: nginx:1.18.0
    ports:
    - containerPort: 80
```

2. Save and exit with `:wq`

3. Ensure a pod is running

```
student@bchd:~$ kubectl apply -f nginx.yaml
```

4. Now check out the logs of the nginx pod.

```
student@bchd:~$ kubectl logs nginx
```

5. It looks like there are no entries. This is because there have been no access attempts to the container. Let's port-forward this pod so that we can use `curl` to get an access attempt logged.

```
student@bchd:~$ kubectl port-forward nginx 2224:80 --address=0.0.0.0
```

6. Make sure you move into a new tmux pane for the following steps. Once there, let's `curl` the NGINX service.

```
student@bchd:~$ curl http://127.0.0.1:2224/
```

```
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
<style>
    body {
        width: 35em;
        margin: 0 auto;
        font-family: Tahoma, Verdana, Arial, sans-serif;
    }
</style>
</head>
<body>
<h1>Welcome to nginx!</h1>
<p>If you see this page, the nginx web server is successfully installed and
working. Further configuration is required.</p>

<p>For online documentation and support please refer to
<a href="http://nginx.org/">nginx.org</a>.<br/>
Commercial support is available at
<a href="http://nginx.com/">nginx.com</a>.</p>

<p><em>Thank you for using nginx.</em></p>
</body>
</html>
```

7. Take a look at the logs of your pod again.

```
student@bchd:~$ kubectl logs nginx

127.0.0.1 - - [16/Apr/2020:03:15:35 +0000] "GET / HTTP/1.1" 200 612 "-" "curl/7.58.0" "-"
```

8. Now we want to stream the logs of your pod. Open a third tmux pane. On the new third pane, run the command below to stream the logs.
```
student@bchd:~$ kubectl logs -f nginx
```

9. On the second terminal, run some commands to produce log output on the first terminal.

```
student@bchd:~$ curl http://127.0.0.1:2224/
```

10. If you really want the log to 'pop', try sending a GET that includes your name in the endpoint request.

```
student@bchd:~$ curl http://127.0.0.1:2224/<your-name>
```

11. **CHALLENGE**: View the logs of a different pod! You may need to launch a new pod to complete this step.
```
SOLUTION step 1 - kubectl apply -f webby-nginx.yaml
SOLUTION step 2 - kubectl logs -f webby-nginx.yaml -c webby # the -c means "the container named webby"
SOLUTION step 3 - split the screen with ctrl+b then SHIFT+double-quote
SOLUTION step 4 - sudo kubectl port-forward webby-nginx 2224:80 # set up a port-forward to your container
SOLUTION step 5 - split the screen AGAIN with ctrl+b then SHIFT+double-quote
SOLUTION step 6 - curl http://127.0.0.1:2224/ # produce traffic to your pod
```

63. Utilize Container Logs

Lab Objective

This lab takes a look at some of the more advanced techniques of logging with and for Kubernetes. This includes basic logging with pods and containers, using a sidecar pod with Fluentd, as well as centralized logging with best practice of separation from the cluster.

Procedure

Basic Logging

1. In this example, we take a look at a pod specification that has two containers writing text to standard out.

```
student@bchd:~$ vim ~/counter-pod.yaml
```

Click here to view the contents of **counter-pod.yaml**

2. While simple on the outside, Kubernetes inherent logging can show you a great deal more than just that. Below is a list of flags for the kubectl logs command. We will utilize some of these going forward.

Name	Shorthand	Default	Usage
all-containers		false	Get all containers' logs in the pod(s)
container	c		Prints the logs of a specified container
follow	f	false	Specify if the logs should be streamed
since		0s	Only return logs new than relative duration
tail		-1	Lines of recent log files to display
timestamps		false	Include timestamps on each line of log output

1. Now that we have the file created, run the pod.

```
student@bchd:~$ kubectl apply -f ~/counter-pod.yaml

pod/counter created
```

2. Verify that the pod has been created.

```
student@bchd:~$ kubectl get pods counter

NAME      READY   STATUS    RESTARTS   AGE
counter   2/2     Running   0          13s
```

3. Let's check what the output is in the logs. Run the following command to try to look at the logs of the Pod we just created.

```
student@bchd:~$ kubectl logs counter

Defaulted container "count" out of: count, countby3
0: Wed Sep 21 16:08:25 UTC 2022
1: Wed Sep 21 16:08:26 UTC 2022
2: Wed Sep 21 16:08:27 UTC 2022
3: Wed Sep 21 16:08:28 UTC 2022
4: Wed Sep 21 16:08:29 UTC 2022
5: Wed Sep 21 16:08:30 UTC 2022
6: Wed Sep 21 16:08:31 UTC 2022
```

> This command used to error in versions earlier than Kubernetes v1.24 -- but now, the **kubectl logs** command will default to the first container specified in the manifest and give the log for that container, while informing the user of this decision.

4. When you have a Pod with more than one container inside of it, you **can** specify which container you wish to view the logs of. Let's try that again with our **count** container.

```
student@bchd:~$ kubectl logs counter -c count
```

```
0: Mon Nov 15 14:57:39 UTC 2021
1: Mon Nov 15 14:57:40 UTC 2021
2: Mon Nov 15 14:57:41 UTC 2021
3: Mon Nov 15 14:57:42 UTC 2021
4: Mon Nov 15 14:57:43 UTC 2021
5: Mon Nov 15 14:57:44 UTC 2021
6: Mon Nov 15 14:57:45 UTC 2021
7: Mon Nov 15 14:57:46 UTC 2021
```

5. Great! Now let's try accessing the logs from the **countby3** container.

```
student@bchd:~$ kubectl logs counter -c countby3

Mon Nov 15 15:02:55 UTC 2021: 105
Mon Nov 15 15:02:58 UTC 2021: 106
Mon Nov 15 15:03:01 UTC 2021: 107
Mon Nov 15 15:03:04 UTC 2021: 108
Mon Nov 15 15:03:07 UTC 2021: 109
Mon Nov 15 15:03:10 UTC 2021: 110
Mon Nov 15 15:03:13 UTC 2021: 111
Mon Nov 15 15:03:16 UTC 2021: 112
```

6. Now, let's try to follow the logs of the **count** container.

```
student@bchd:~$ kubectl logs counter -c count -f

444: Mon Nov 15 15:05:04 UTC 2021
445: Mon Nov 15 15:05:05 UTC 2021
446: Mon Nov 15 15:05:06 UTC 2021
447: Mon Nov 15 15:05:07 UTC 2021
448: Mon Nov 15 15:05:08 UTC 2021
449: Mon Nov 15 15:05:09 UTC 2021
450: Mon Nov 15 15:05:10 UTC 2021
451: Mon Nov 15 15:05:11 UTC 2021
```

> To exit the logs, do a `Ctrlc` since you used the option of **-f** to "follow" them.

7. Mini Challenge: Follow the logs of the **countby3** container.

> You got this!

8. Now let's try to get the logs from all of our containers in this Pod.

```
student@bchd:~$ kubectl logs counter --all-containers

Mon Nov 15 15:06:46 UTC 2021: 182
Mon Nov 15 15:06:49 UTC 2021: 183
Mon Nov 15 15:06:52 UTC 2021: 184
Mon Nov 15 15:06:55 UTC 2021: 185
Mon Nov 15 15:06:58 UTC 2021: 186
Mon Nov 15 15:07:01 UTC 2021: 187
Mon Nov 15 15:07:04 UTC 2021: 188
Mon Nov 15 15:07:07 UTC 2021: 189
```

> Wait, this looks like we just got the logs from the **countby3** container! That is because the logs are read from the first container first, and the second container second, not mixed together. If you look back through the history of the output or use a tool like **less**, you can view all of them.

9. Another way to look at some of the logs from all of our containers is to specify a **since** flag. Let's see what has happened in both of our containers for the last 10 seconds.

```
student@bchd:~$ kubectl logs counter --all-containers --since 10s
```

```
799: Mon Nov 15 15:10:59 UTC 2021
800: Mon Nov 15 15:11:00 UTC 2021
801: Mon Nov 15 15:11:01 UTC 2021
802: Mon Nov 15 15:11:02 UTC 2021
803: Mon Nov 15 15:11:03 UTC 2021
804: Mon Nov 15 15:11:04 UTC 2021
805: Mon Nov 15 15:11:05 UTC 2021
806: Mon Nov 15 15:11:06 UTC 2021
807: Mon Nov 15 15:11:07 UTC 2021
808: Mon Nov 15 15:11:08 UTC 2021
Mon Nov 15 15:11:01 UTC 2021: 267
Mon Nov 15 15:11:04 UTC 2021: 268
Mon Nov 15 15:11:07 UTC 2021: 269
```

10. Or if we want to just look at the last 5 lines of logs for all of our containers, we can use the **tail** flag.

```
student@bchd:~$ kubectl logs counter --all-containers --tail 5

932: Mon Nov 15 15:13:13 UTC 2021
933: Mon Nov 15 15:13:14 UTC 2021
934: Mon Nov 15 15:13:15 UTC 2021
935: Mon Nov 15 15:13:16 UTC 2021
936: Mon Nov 15 15:13:17 UTC 2021
Mon Nov 15 15:13:04 UTC 2021: 308
Mon Nov 15 15:13:07 UTC 2021: 309
Mon Nov 15 15:13:10 UTC 2021: 310
Mon Nov 15 15:13:13 UTC 2021: 311
Mon Nov 15 15:13:16 UTC 2021: 312
```

11. Although our logs already have timestamps on them, let's also see how the kubectl logs flag of **timestamps** is able to apply their own.

```
student@bchd:~$ kubectl logs counter --all-containers --tail 5 --timestamps

2021-11-15T15:16:50.488034356Z 1149: Mon Nov 15 15:16:50 UTC 2021
2021-11-15T15:16:51.490242494Z 1150: Mon Nov 15 15:16:51 UTC 2021
2021-11-15T15:16:52.491539522Z 1151: Mon Nov 15 15:16:52 UTC 2021
2021-11-15T15:16:53.492712346Z 1152: Mon Nov 15 15:16:53 UTC 2021
2021-11-15T15:16:54.494665484Z 1153: Mon Nov 15 15:16:54 UTC 2021
2021-11-15T15:16:41.067750117Z Mon Nov 15 15:16:41 UTC 2021: 380
2021-11-15T15:16:44.069552746Z Mon Nov 15 15:16:44 UTC 2021: 381
2021-11-15T15:16:47.071627156Z Mon Nov 15 15:16:47 UTC 2021: 382
2021-11-15T15:16:50.072944094Z Mon Nov 15 15:16:50 UTC 2021: 383
2021-11-15T15:16:53.074621152Z Mon Nov 15 15:16:53 UTC 2021: 384
```

12. **Great Job!** You have just tried out all of the various flags that kubectl logs allows you to use. Now take a few minutes and feel free to try grabbing the logs from them as well.

13. Before moving on to our next topic, please remove this logging Pod.

```
student@bchd:~$ kubectl delete -f ~/counter-pod.yaml
```

Click for Code to Catch Up!

```
wget https://labs.alta3.com/courses/kubernetes/files/counter-pod.yaml -O ~/counter-pod.yaml
kubectl apply -f ~/counter-pod.yaml
sleep 10
kubectl get pods counter
kubectl logs counter
kubectl logs counter -c count
kubectl logs counter -c countby3
kubectl logs counter -c count -f > /dev/null 2>&1 &
kubectl logs counter --all-containers
kubectl logs counter --all-containers --since 10s
kubectl logs counter --all-containers --tail 5
kubectl logs counter --all-containers --tail 5 --timestamps
kubectl delete -f ~/counter-pod.yaml
```