Kubernetes

מבוא:

K8S הוא הכלי הנפוץ ביותר לניהול קונטיינרים. הוא **לא** מהווה תחליף ל-docker engine.

קיימים ל-K8S מספר יתרונות לניהול הקונטיינרים:

* מכיל load balancer
* בעל יכולת אחסון עצמאית כגון: SAN, NAS EBS וכו'
* rollout ו-rollback אוטומטיים
* self healing
* ניתן לנהל את הקונפיגורציה בצורה של volumes ו-variables
* ועוד ועוד

K8S Set Up Tools:

* **manual way** - לא מומלץ, זוהי דרך קשה להתקנה. מומלץ להשתמש בה למטרות למידה בלבד.
* **minikube** - יוצר NODE אחד בלבד על המחשב האישי שירוץ על oracle VM - מומלץ למטרות למידה ו-testing
* **kubeadm** - דרך יעילה יותר מה-2 הקודמות, מומלת לשימוש ב-production. יוצר multi node לפי בקשתינו. ניתן ליישם אותו על EC2, מחשב פיזי, VM ועוד
* **kops** - הדרך המומלצת ביותר והיעילה ביותר להתקנה. נעשה בה שימוש בעיקר מול AWS אך אפשרי ליישם אותה בעוד פלטפורמות.

פקודות לניהול K8S:

אנו ננהל את ה-cluster שלנו באמצעות הפקודה kubectl

נוכל ליצור, למחוק ולערוך אובייקטים ב-cluster באמצעות הפקודה הזאת.

מצורף [קובץ PDF עם cheat sheet](https://drive.google.com/file/d/1OivfP4mXZtVNyt7FiWzYm1bSxWmkfJTO/view?usp=drive_link) של כלל האפשרויות של הפקודה.

ארכיטקטורה:

**Master Node -** זהו הרכיב השולט ב-worker nodes, המלך של ה-K8S. המכיל מספר רכיבים בתוכו.

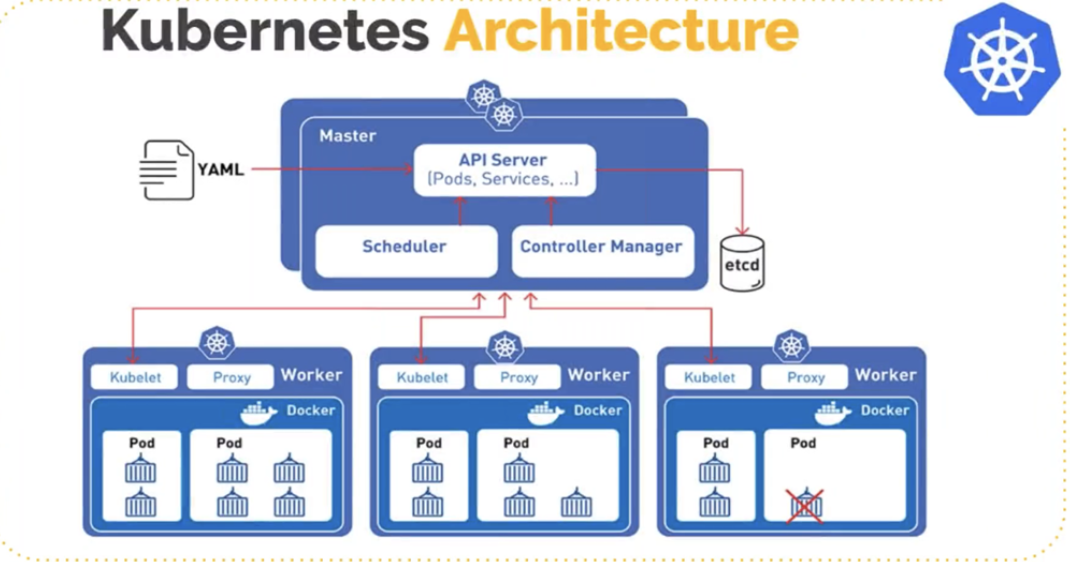
* **API Server -** הוא מטפל בכלל הבקשות היוצאות והנכנסות מה-nodes, ובכלל התקשורת בין ה-services. מהווה כ-front-end של ה-control plane. ניתן להתחבר אליו כ-ADMIN באמצעות kubectl CLI. בנוסף אליו, קיימת גם דרך webית בה ניתן לגשת אל ניהול ה-cluster.
* **ETCD Storage** - מכיל את כל המידע של ה-kubernetes cluster - את המצב הנוכחי שלו. ה-API Server ניגש אל ה-ETCD באופן סדיר כדי לשלוף מידע. מומלץ לגבות אחסון זה באופן תדיר.
* **Scheduler** - מתזמן את הקונטיינרים כדי שירוצו על גבי ה-NODE הנכון. מה הכוונה? הוא מצפה לבקשה, ברגע שהוא מקבל אותה, הוא מעביר את המידע הלאה אל ה-worker node הרלוונטי ומריץ את הקונטיינר עליו. ההחלטה איפה להריץ את הקונטיינר תתבסס לפי מספר נתונים: מה המשאבים הדרושים לקונטיינר הספציפי הזה, מה מצב החומרה והתוכנה על גבי ה-NODES הקיימים ועוד…
* **Controller Manager** - למעשה זהו רכיב בעל מספר תפקידים, אשר רוכזו לרכיב אחד בודד

ה-controller מכיל:

* node controller - מנטר את ה-worker nodes.
* replication controller - מנטר את ה-PODS ומבצע auto healing במקרה הצורך.
* endpoint controller - מאכלס את כלל ה-endpoint objects.
* service account & token controller - מנהל את ההרשאות ב-cluster.

**Worker Node** -שרת עליו מותקן docker engine ורצים עליו הקונטיינרים הרלוונטים. מכיל את הרכיבים הבאים:

* **kubelet** - זהו agent שרץ בכל NODE - תפקידו להאזין לבקשות או הפקודות של ה-MASTER. ברגע שה-scheduler מחליט ש-worker node ספציפי יהיה אחראי להרצת קונטיינר כלשהו, הוא למעשה מעביר את האחריות הזו ל-kubelet. הוא אחראי להביא את ה-IMAGE הרלוונטי ולעשות למעשה את "העבודה הכבדה" בשבילנו.
* **kube proxy** - רכיב זה מהווה את ה-NETWORK בתוך ה-NODE. כמו כל רשת אחרת, גם לו יש network rules, כמו שיוך של security group.
* **container runtime** - יכול להיות איזה סוג שנבחר (כאשר הנפוץ ביותר זה DOCKER)
* **pods** - הרכיב אשר נותן את המשאבים הדרושים להרצת הקונטיינר אשר רץ בתוכו (ניתן להריץ יותר מקונטיינר אחד ב-POD). יתר על כן, לכל POD יש IP address ייחודי משלו.



Pods:

הינו היחידה הקטנה והבסיסית ביותר ב-K8S cluster, והיא מכילה אוסף של קונטיינר אחד או יותר. pod מייצג process שרץ ב-cluster שלנו.

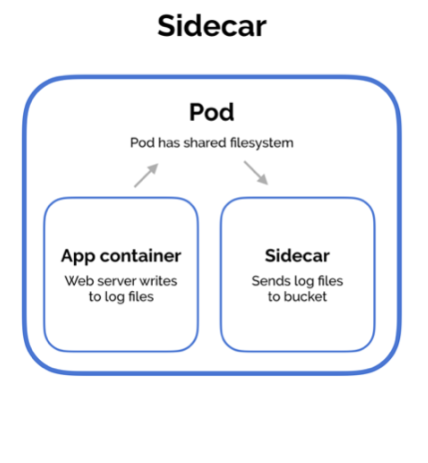
סוגי קונטיינרים ב-POD:

* **init** - קונטיינר זמני, מטרתו העיקרית היא לבצע פעולות מסוימות הנדרשות לנו בתחילת ה-POD, ולאחר מכן ה-init container יימחק מה-POD. לאחר מכן, יווצר ה-main container.
* **sidecar** - קונטיינר שמטרתו לעזור ל-main container לפעול כמו שצריך. לדוגמא: קונטיינר האחראי לאיסוף לוגים של הראשי, או קונטיינר האחראי לניטור הראשי.
* **main** - הקונטיינר הראשי ב-POD, הוא יכיל את האפליקציה/המערכת שנרצה שתרוץ עליו.

ארכיטקטורה:

* **Pod that run a single container** - זוהי שיטה המיישמת את מודל: "one container per pod". זוהי הארכיטקטורה האידיאלית ליציאת pods. הוא מהווה מעטפת לקונטיינר יחיד, למעשה K8S cluster מנהל את ה-pods ישירות, ולא את הקונטיינר.
* **Multi container pod** - כלל הקונטיינרים תחת ה-pod יחלקו את אותם המשאבים. לפי ארכיטקטורה זו, ב-POD יחיד יהיה לנו קונטיינר ראשי, וקונטיינר משני מסוג sidecar או init שמטרתו לסייע ל-POD הראשי. לדוגמא: יהיה לנו pod שמכיל קונטיינר ראשי עם אפליקציה וקונטיינר משני שמטרתו להכיל את כלל הלוגים של האפליקציה הראשית.

כל POD נועד להריץ אפליקציה אחת בלבד. אם נרצה ליישם HA, נייצר מספר PODים עם אותן האפליקציות, ולא מספר קונטיינרים בתוך POD אחד.



YAML configuration file:

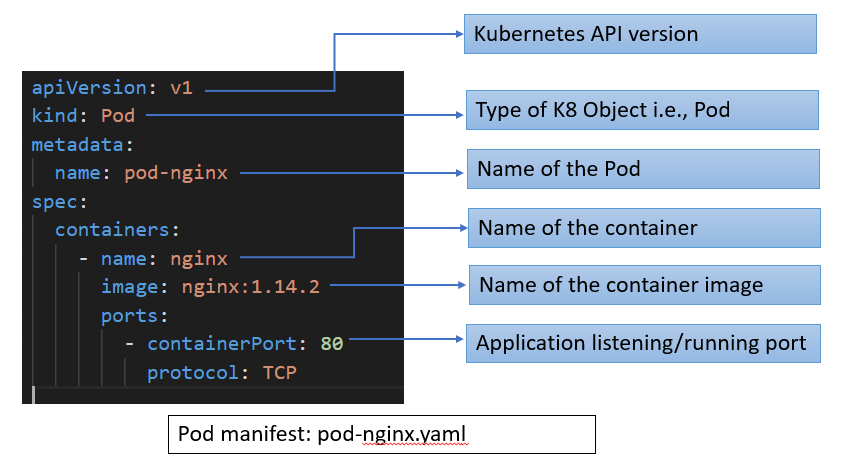
ניתן להגדיר POD באמצעות קובץ YAML (יישום של IAAS).

קיימות כמה הגדרות חובה בקובץ ה-YAML שנהיה חייבים ליישם.

* **apiVersion** - הגרסה של K8S, נוכל לדעת מה הגרסה הרלוונטית באמצעות בדיקה בדוקומנטציה.
* **kind** - איזה סוג של אובייקט נרצה ליצור, במקרה שלנו מדובר ב-pod.
* **metadata -** אינפורמציה על ה-POD. ערך זה מצויין כ-dictionary בקובץ ה-YAML.
* **spec** - מידע טכני על ה-POD, כמו שם הקונטיינר וה-image בו נעשה שימוש. גם מצויין כ-dictionary.

נוכל ליצור POD מ-YAML באמצעות הפקודה הבאה:

kubectl create -f *<pod\_setup\_yaml>*



network:

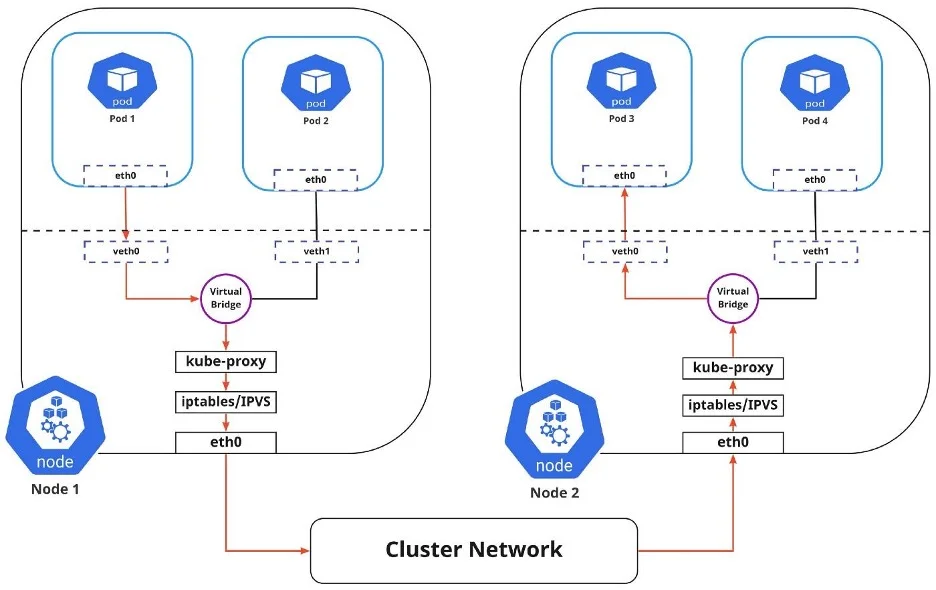
ניקח לדוגמא מצב, כמו בתמונה שלפנינו, ש-POD 1 רוצה לתקשר עם POD 4.

כיצד זה יתבצע?

לכל NODE קיימת רשת פרטית משלו, נוסף לרשת הפרטית קיים overlay network המקשרת בין ה-NODEים.

כל POD יכול לתקשר עם ה-PODים האחרים בתוך ה-NODE באמצעות **bridge**.

על מנת לבצע תקשורת מחוץ ל-NODE, ה-bridge ייגש ל"ראוטר", כלומר ל-eth0 ובך יוכל לתקשר עם ה-NODEים האחרים שנמצאים ב-overlay network.



Kubeconfig File:

כאשר אנו מתחברים ל-cluster באמצעות הפקודה kubectl, איך היא יודעת לאיזה cluster בדיוק להתחבר? ועם אילו פרטים? ההתחברות הזו נעשית באמצעות ה-kubeconfig file. זהו קובץ yaml אשר מכיל את כלל הפרטים הדרושים להתחברות ל-cluster שלנו.

היכן מאוחסן הקובץ הזה? ברגע שניצור cluster, תיווצר בתיקיית ה-home של המשתמש איתו יצרנו את ה-cluster תיקיה בשם **kube.** ובתוכה יהיה הקובץ **config**

אילו פרטים מכיל קובץ זה?

**cluster** - תחת טאב זה יופיעו כלל הפרטים הרלוונטים ל-cluster. יכולים להיות מספר clusters בקובץ config אחד

* certificate - הסרטיפיקט של ה-cluster
* server - זהו ה-URL של ה-master node - כלומר של ה-API server
* name - שם ה-cluster

**users** - תחת טאב זה יופיעו כלל הפרטים הרלוונטים למשתמשים אשר מתחברים ל-cluster

* name - שם המשתמש
* certificate - הסרטיפיקט של המשתמש
* key - נעשה בו שימוש על מנת ליצור authentication למשתמש

**context** - טאב זה משייך בין cluster לבין user

* cluster - שם ה-cluster
* user - שם המשתמש שמשתייך ל-cluster

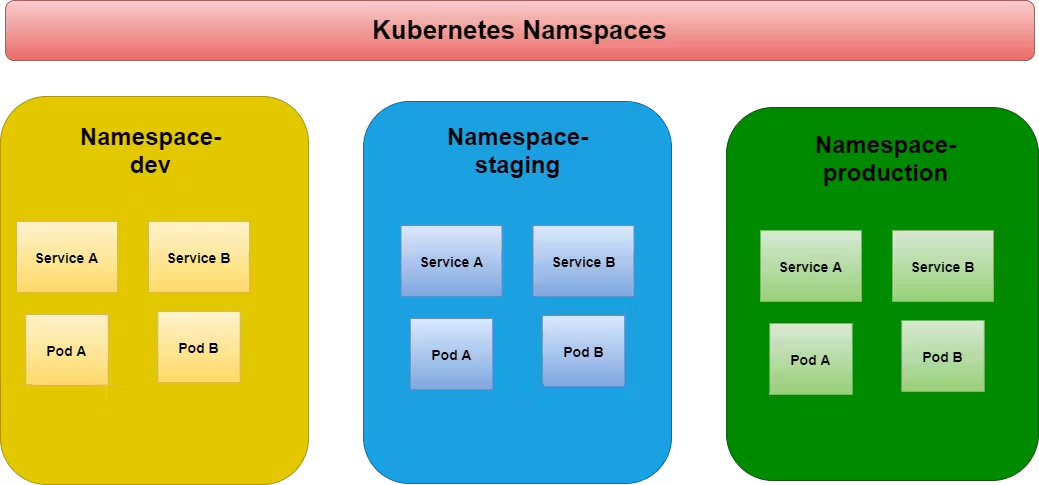
**current context** - ערך זה יכול להופיע פעם אחת בלבד, לא כמו שאר הערכים אשר יכולים להופיע מספר פעמים (יכול להיות מספר של clusters או מספר של users). ערך זה מכיל את ה-cluster הנוכחי אליו תתבצע ההתחברות באמצעות הפקודה kubectl. (ניתן לשנות את ה-cluster אליו נרצה להתחבר בעת הרצת הפקודה, באמצעות הוספת variables מתאימים)



Namespaces:

רכיב ב-cluster היוצר בידוד בין קבוצות של אובייקטים בתוך cluster יחיד, הוא למעשה מאגד קבוצת אובייקטים ב-cluster. כל שם של אובייקט ב-namespace חייב להיות ייחודי, אך בין NSים אין צורך בכך.

מה הכוונה? לדוגמא, ב-NS1 קיים לי POD בשם NGNIX1, לא ניתן ליצור עוד אובייקט בשם זה. אבל, ב-NS2 ניתן ליצור POD בשם NGNIX1.



באופן דיפולטי, בעת יציאת k8s cluster, נוצרים שלושה Namespaces:

**default** - נוצר באופן אוטומטי על מנת שנוכל להכיל את הפרויקט שלנו עליו, בלי צורך ליצור NS חדש.

**kube-system** - מכיל באופן אוטומטי PODS הקשורים לניהול ה-cluster, לדוגמא: etcd, scheduler וכו'

**kube-public** - סוג זה קריא לכל ה-clients, לכן נוכל לאחסן בו אובייקטים שרצה שיהיו חשופים לכל.

נוסף אליהם, אנו תמיד יכולים ליצור עוד namespaces, ולבצע את חלוקת האובייקטים לפי בחירתינו.

מתי נרצה להשתמש בזה בכלל? בעת חלוקה ל-DEV, TEST, PROD או אם נרצה לחלק את ה-NS לפי פרוייקטים.

Volumes:

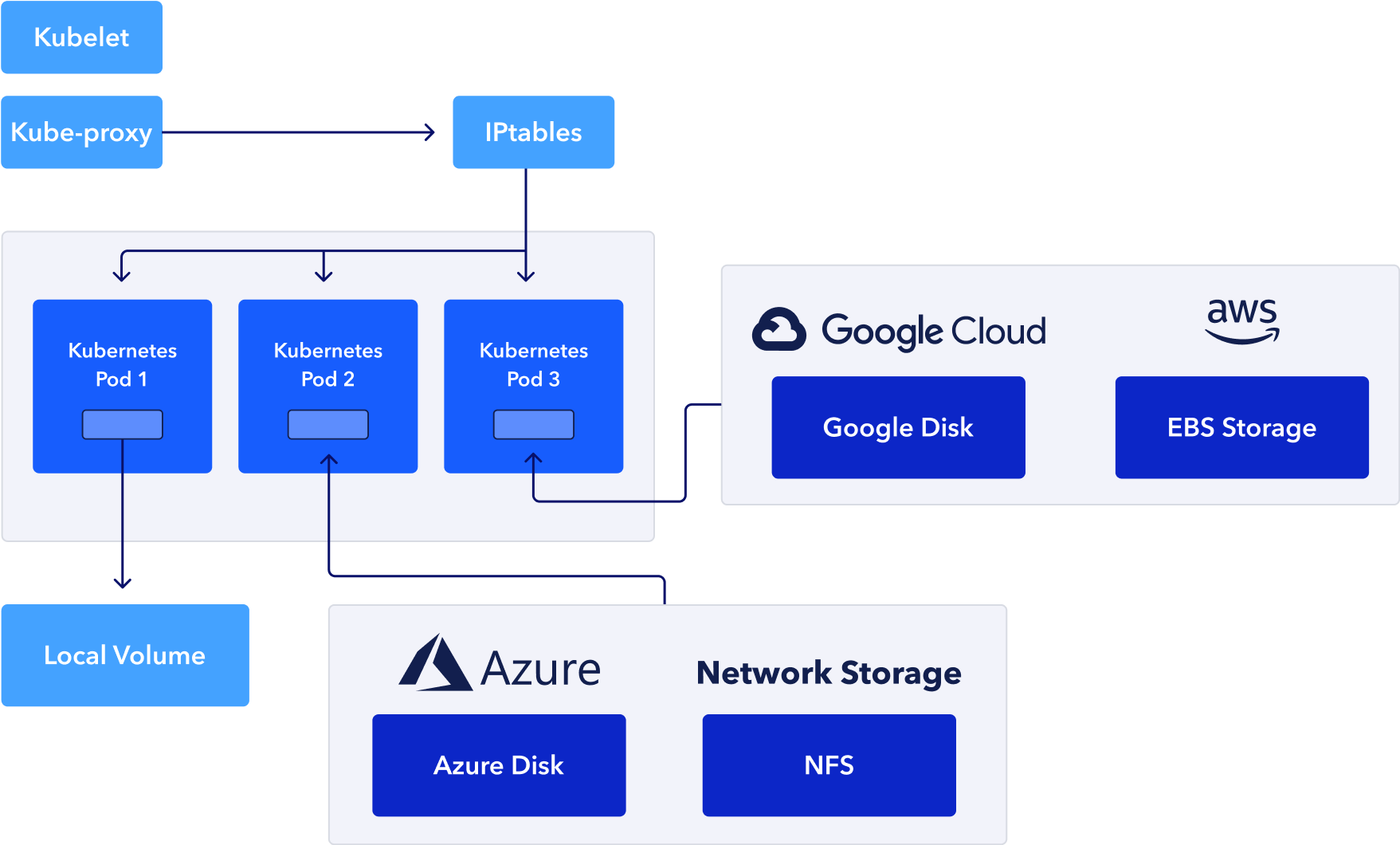
זהו אובייקט אשר מייצג אחסון של קונטיינר ב-cluster שלנו.

קיימים המון סוגי אחסון מהם נוכל ליצור volume לדוגמא: EBS, GlusterFS, NFS, local ועוד…

בקובץ ה-YAML של ה-POD שלנו, אנו נציין את ה-mount point הרלוונטי, כלומר את הנתיב בתוך הקונטיינר אותו נרצה לגבות, שהמידע שלו יישמר במקור האחסון שנבחר לו.

בנוסף נציין את ה-volume - הנתיב הרלוונטי בו יישמר האחסון במקור.

[מצורף PDF](https://drive.google.com/file/d/1NTWzXgHoNCdrGZuGzUfycsUM5u3mt-WM/view?usp=drive_link) עם כלל סוגי האחסון האפשריים והדרך בה מקנפגים אותם.



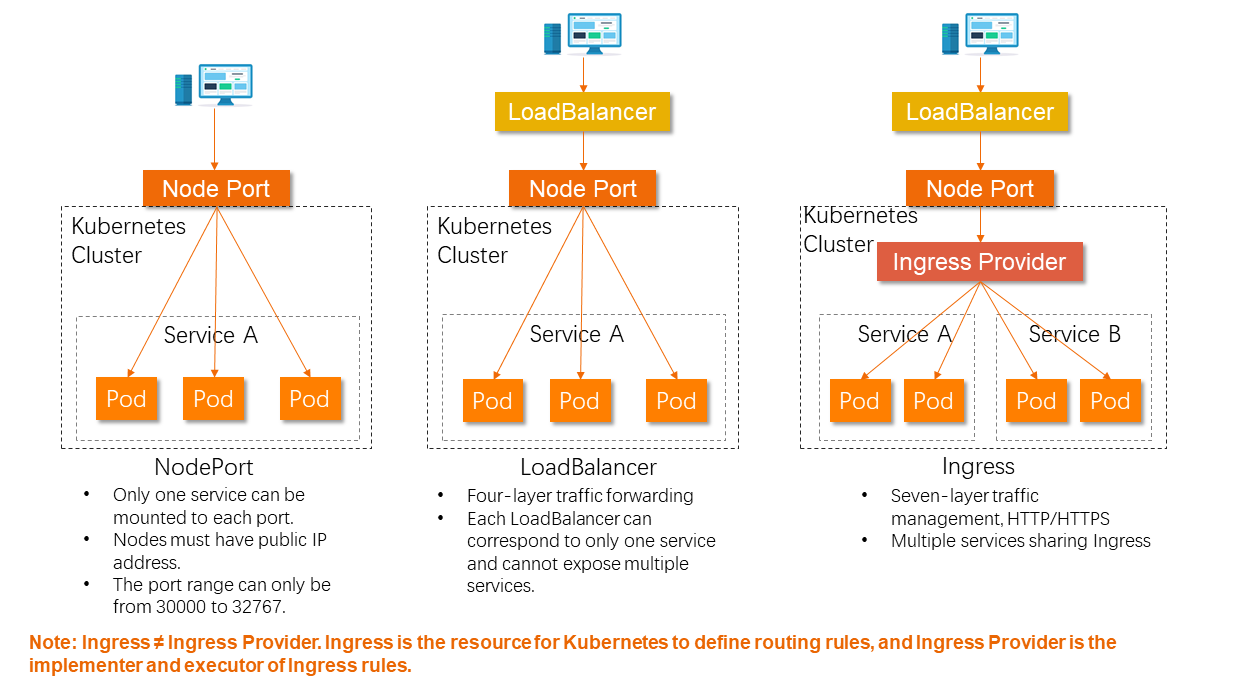
Services:

זהו אובייקט ב-K8S שמטרתו לחשוף PODS, או יותר נכון את האפליקציה שתחת ה-POD כ-application service.

אבל, ביצירה של POD ניתן להגדיר לו פורט לחשיפה ולעשות בו שימוש. אז למה בכלל צריך service?

POD הינו אובייקט זמני וחד פעמי - ברגע שנרצה לבצע בו שינוי כלשהו, נצטרך למחוק אותו ולהגדירו מחדש. כך שגם כתובת ה-IP שלו היא זמנית. כאן נכנס ה-service.

זהו אובייקט **סטטי** אשר נותן לנו מעין נקודת קצה, המקשרת אליו ה-PODS ונותנת לנו את היכולת לחשוף אותם בקלות.

סוגי services:

* **Node port** - נשתמש בסוג זה למטרות TEST בלבד. הוא מאוד פשוט: לוקח פורט וחושף אותו כלפי ה-NODE.

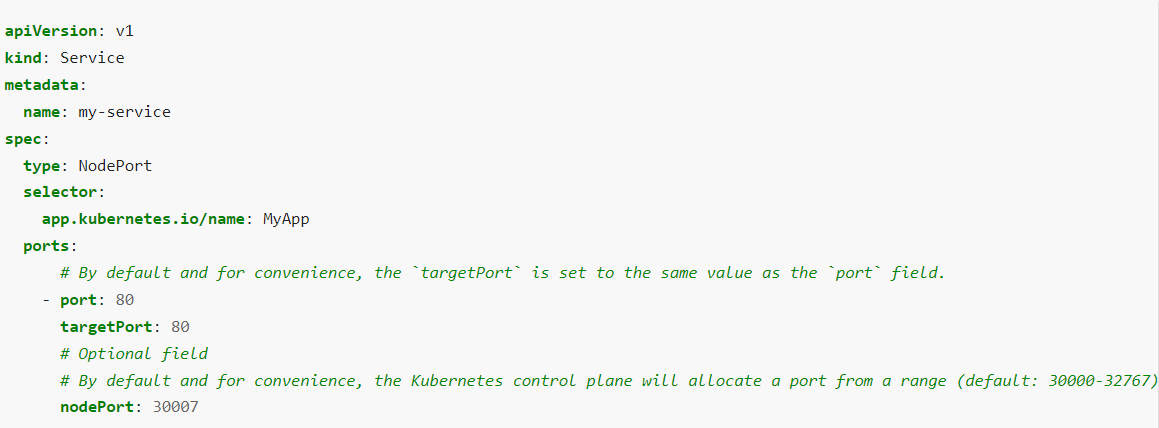
הוא מכיל backend port - זהו הפורט אליו משוייכים הקונטיינרים. וגם frontend port - המיועד לתקשורת פנימית ב-NODE.

כיצד הוא יודע לאיזה PODS עליו לגשת? הרי בתוך NODE עשויים להיות מאות של PODS.

הוא עושה זאת באמצעות ה-label selector. אנו נקצה לו שם של LABEL כלשהו, וכל POD אשר יש לו את אותו LABEL ינוטב אל ה-service הזה.

בנוסף אנו נציין לו את ה-node port, שזה הפורט שיוקצה ל-NODE וייחשף כלפי חוץ.

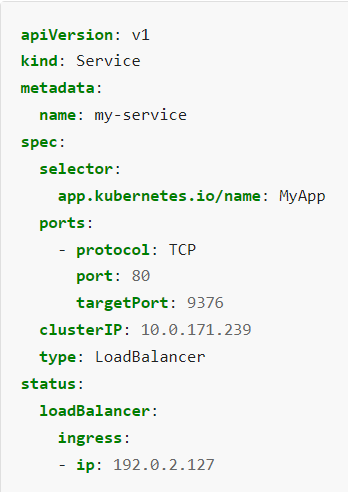
לכן, ברגע שנציין את ה-IP של ה-NODE עם ה-node port, הוא באופן אוטומטי ייגש ל-service שייגש ל-POD שייגש לקונטיינר ויחשוף אותו כלפי חוץ.



target port - ההגדרה של ה-backend port.

node port - זהו הפורט המיועד לתקשורת חיצונית.

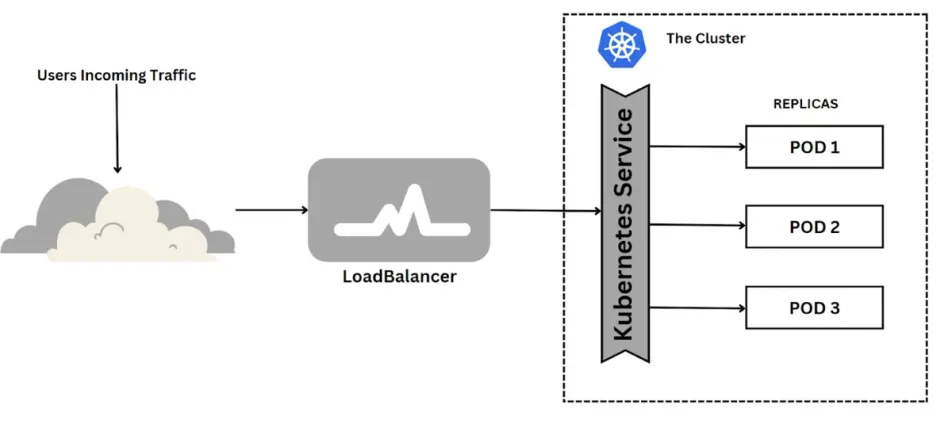
selector - שם ה-LABEL לפיו יבחרו ה-PODS הרלוונטים.

* **Cluster IP** - משמש לתעבורה פנימית של מידע ב-NODE. למשל אם ה-backend צריך לתקשר עם ה-front וכו'.
* **Load Balancer** - אנו נשתמש בסוג זה למטרות production. הוא למעשה ממש load balancer עם DNS המוקצה לו, שנמצא שלב אחד מעל ה-nodes (ב-AWS הוא גם ממש יוצר ELB). בעת יצירת load balancer נוצר לנו גם באופן אוטומטי node port המוקצה לו מספר פורט באופן רנדומלי.

אז איך LB עובד בעצם? load balancer ניגש לאחד ה-NODES באמצעות ה-node port, והוא ממפה את הבקשה אל הקונטיינר הרלוונטי.

port - מספר הפורט של ה-load balancer

target port - מספר הפורט של הקונטיינרים



Ingress:

זהו אובייקט API שמטרתו לנהל את הגישה ל-service שנבחר ב-cluster.

הוא מנטב תעבורת HTTP\HTTPS מהרשת שנמצאת מחוץ ל-cluster אל ה-service בתוך ה-cluster, הוא למעשה משמש ממש כ-**load balancer** לניתוב התעבורה.

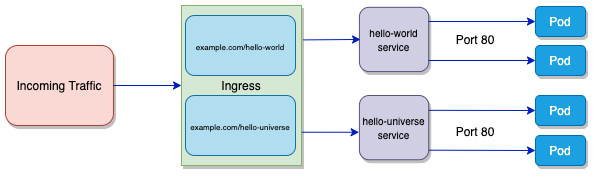
על מנת שנוכל ליצור ingress, נהיה חייבים ליצור קודם לכן **ingress controller.**

(מה הכוונה? כמו של-K8S קיים controller, כך גם ל-ingress. רק שהוא לא מגיע בצורה מובנית יחד עם ה-ingress, אלא אנו מתקינים אותו באופן עצמאי ונפרד. סוג ה-ingress הוא לחלוטין לבחירינו)

קיימים המון סוגים של ingress control כאשר הנפוץ ביותר הוא: Nginx.

לדוגמא ב-K8S שמותקן על סביבת AWS, ה-ingress controller יהיה למעשה (NLB (network load balancer.

Ingress מכיל אוסף של חוקים לפיהם תנותב התעבורה.



בתמונה הנוכחית ניתן לראות קובץ YAML של ingress

**ingressClassName** - שם ה-controller שנעשה בו שימוש

**rules** - רשימת החוקים של ה-ingress

**path** - הנתיב ב-URL אליו תנותב התעבורה

**host** - בדרך כלל ניתן להוסיף תחת rules גם אופציה של host, שהוא מציין מאיזה DNS תנותב התעבורה.

בדוגמא שכאן לא מצויין host, לכן התעבורה תעבור דרך כל כתובת ה-IP ב-cluster.

**backend** - שילוב של שם ה-service והפורט אליו הוא מפנה

**תזכורת: אנו מגדירים ingress ל-service ולא ל-POD**

ReplicaSet:

זהו אובייקט אשר אחראי על יצירת שכפולים ל-PODS ב-cluster שלנו. מדוע יש בו צורך?

ניקח לדוגמא מצב בו יש לנו NODE המכיל POD כלשהו אליו ניגשים יוזרים חיצוניים. אם ה-POD נכשל, או שיש לו שגיאה כלשהי, אז אנחנו נצטרך למחוק אותו וליצור אותו מחדש באופן ידני.

לשם כך נוצר ה-ReplicSet - הוא יוצר לנו סקלביליות ל-PODים ב-NODE שלנו.

כאשר נציין מספר ReplicaSets, אזי ה-scheduler יכול להפיץ אותם אל מספר worker nodes. במצב זה גם אם כלל ה-node נהרס, הוא יכול ליצור את ה-PODS שרצו עליו על גבי NODE אחר.

כיצד נוכל לשנות את הסקלביליות שהוגדרה ב-RS?

לשם כך קיימות שלושה דרכים:

1. שינוי באופן ידני של קובץ ה-YAML, ולאחר מכן הרצת הפקודה הבאה:

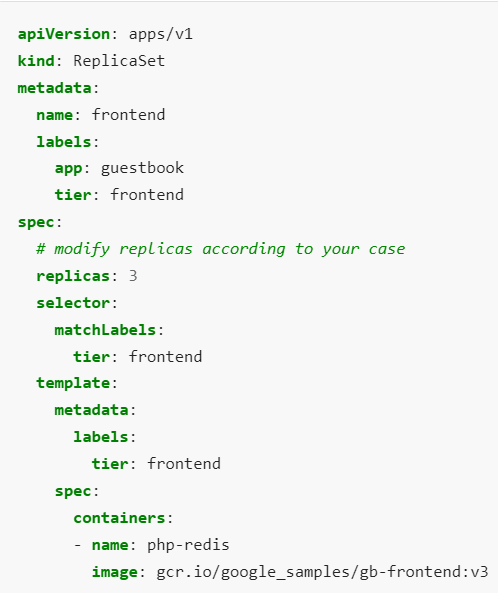
kubectl apply -f *<ReplicaSet\_YAML>*

1. הרצת הפקודה הבאה:

kubectl scale --replicas=*<number\_of\_replicas>* rs/*<ReplicaSet\_name>*

1. שינוי ההגדרה באמצעות EDIT:

kubectl edit rs *<ReplicaSet\_name>*



**replicas -** מספר השכפולים שיווצרו ל-POD.

**selector** - מציין את ה- LABEL של ה- POD שייבחר, אותו יש לשכפל. לדוגמא: אם יצרתי RepliceSet שמוגדר לו ליצור 3 שכפולים, וקיים לי כבר POD אחד עם ה-LABEL שצוין. אז הוא יצור לי רק עוד 2.

**spec** - התנאים הטכניים ליצירת ה-POD כמו: image, name וכו'.

Deployments:

זהו אובייקט ב-cluster שאחראי על יצירת שינויים ב-POD.

ניקח לדוגמא מצב בו יש לנו POD, ואנו רוצים לשדרג את ה-image שלו לאחד יותר חדש, אנו נצטרך למחוק אותו, לערוך באופן ידני את קובץ ה-YAML, ולהעלות את ה-POD מחדש.

deployment עושה עבורנו את כל התהליך הזה **בצורה אוטומטית**, בנוסף, הוא יכול לעשות את זה עבור ReplicaSet, ולא רק ל-POD יחיד (למעשה, קובץ ה-YAML שלו מאוד דומה לזה של ReplicaSet).

בנוסף לכך שניתן לשדרג images של PODS בקלות, deployment מאפשר לנו לעשות **rollback**.

deployment גם מאפשר לנו יצירת סקלביליות: נוכל לשלוט בקלות בכמות ה-PODים שרצים תחת ה-deploy - להגדיל ולקטין לפי רצוננו.

על מנת שנוכל לבצע שינויים ב-PODים שלנו, נוכל פשוט לשנות את ההגדרה של ה-deployment.

למשל, אם נרצה לשדרג את ה-image הנוכחי לגרסה גבוהה יותר, קיימות מספר דרכים אפשריות:

1. נערוך באופן ידני את קובץ ה-YAML של ה-deployment, ונעשה לו apply.
2. נריץ את הפקודה הבאה:

kubectl set image deployment/<deploy\_name> <container\_name>=<image>:<tag>

אם נרצה לבצע rollback לגרסה הקודמת, נריץ את הפקודה הבאה:

kubectl rollout undo deployment/<deploy\_name>

אבל איך rollback עובד בעצם? איך הוא יודע לחזור לגרסה הקודמת שהייתה לו?

* כשאנו יוצרים deployment נוצר יחד איתו באופן אוטומטי **ReplicaSet**, עם כלל הפרטים הרלוונטים.

ברגע שנרצה לשדרג גרסה של image, ה-deploy לוקח את ה-RS הישן, ומשנה את מספר הקונטיינרים שרצים בו ל-0. בנוסף הוא יוצר RS חדש עם ה-image המשודרג, ובו מספר ה-PODים שרצים הוא המספר שבחרנו.

כאשר אנו נרצה לעשות rollback, הוא יאפס את מספר ה-PODים ב-RS המשודרג, ויעלה את מספר ה-PODים ב-RS הקודם. ברגע שנריץ את פקודת ה-rollback, נוכל לציין בה גם בדיוק לאיזו גרסה נרצה לחזור.

לאחר שדרוג של image, כאשר נריץ את הפקודה: kubectl get rs יתקבל הפלט הבא:

NAME DESIRED CURRENT READY AGE

nginx-deployment-new 3 3 3 25s

nginx-deployment-old 0 0 0 36s

כלומר: יש לנו שלושה PODים שרצים על גבי ה-deployment החדש, ואפס שרצים על גבי הישן.

Config Map:

ננסה לדמיין לנו מצב בו יש לנו POD, המכיל קונטיינר בו קיימים 20 environment variables.

ברגע שנרצה לבצע שינוי כלשהו בקונטיינר הנ"ל, נצטרך למחוק וליצור אותו מחדש, כולל את כל ה-environment variables הללו.

ונגיד שה-variables הללו צריכים להיות קיימים בעוד עשרה קונטיינרים נוספים - קבצי ה-YAML שלהם יהיו גדולים ומסורבלים, ובכל פעם שנרצה לבצע שינוי נצטרך ליצור אותם מחדש.

לשם כך נוצר ה-config map - זהו אובייקט המכיל רשימה של environment variables, אותם ניתן לשייך **לקונטיינר** (שימו לב - ברמת הקונטיינר, לא ברמת ה-POD)

בתמונה המצורפת נוכל לראות דוגמא לקובץ YAML של config map, בו קיימים ארבעה variables.

תחת metadata נציין את הפרטים הטכניים של ה-CM, כגון שמו.

תחת data נציין את פרטי ה-variables.

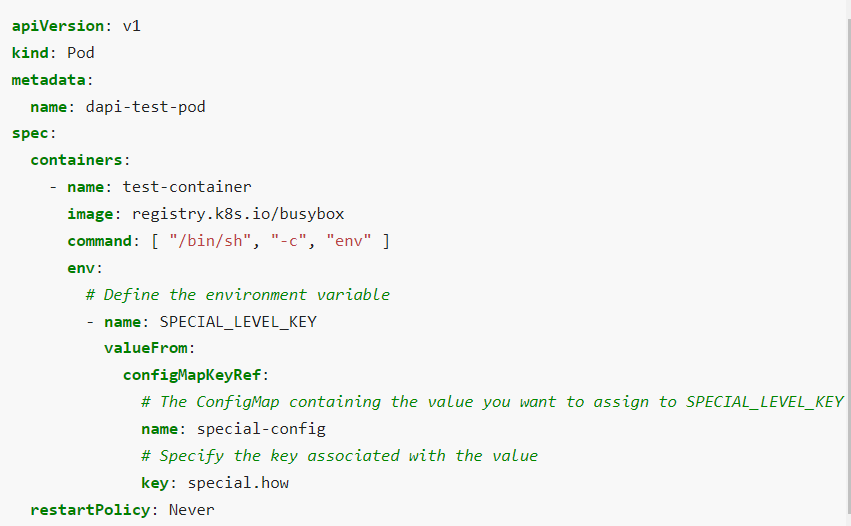
כיצד אפשר לשייך את ה-CM ל-POD?

לשם כך קיימות שלוש דרכים:

1. **ברמת הקונטיינר באמצעות envFrom:**

בתמונה מספר 1 ניתן לראות שהוסיפו envFrom ותחתיו נתנו את שם ה-CM.

במצב זה, כלל ה-variables שרשומים ב-CM יופיעו בקונטיינר.

1. **ברמת הקונטיינר באמצעות valueFrom:**

בתמונה מספר 2 ניתן לראות שתחת הערך env ציינו את שם ה-variable הרצוי.

ותחת הערך valueFrom ציינו את ה-CM ממנו ירצו לקחת את המידע, ובנוסף את שם ה-key.

מה זה key? זהו הערך ב-config map ממנו יילקח המידע.

כעת נוצר מצב בו הערך של ה-key הוא הערך של ה-name.

1. **באמצעות יצירת readonly volume** -

בתמונה מספר 3 ניתן לראות שיצרו volume

בשם config-volume והוא מכיל את

ה-variables של ה-CM המצוין.

קיימת אופציה לציין רק variables

מסוימים ולא את כלל ה-CM.

Secrets:

בדומה ל-config map, זהו אובייקט ב-cluster שמטרתו לאחסן variables, רק שבשונה מ-CM, הוא **מצפין** אותם.

אנו נשתמש ב-secret כאשר נרצה לאחסן נתונים רגישים כמו: סיסמאות, שם משתמש, שמות שרתים, וכו'.

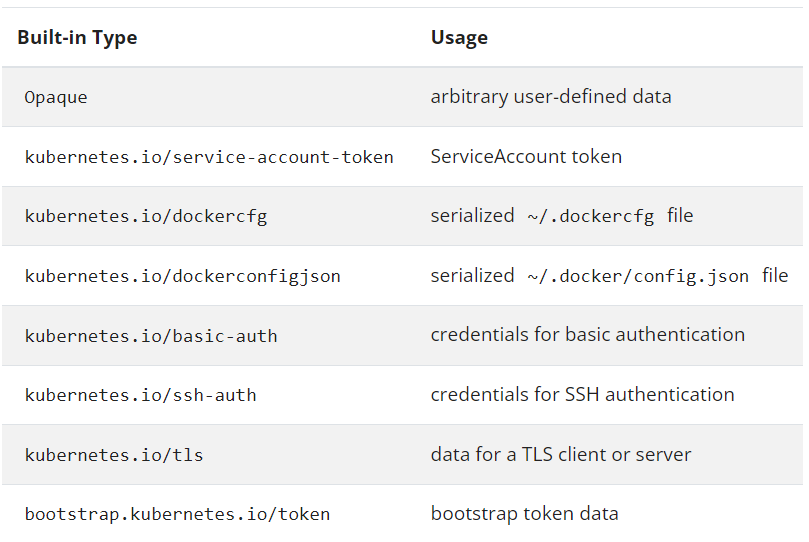
השימוש הנפוץ ביותר ל-secret הוא אחסון פרטי ההתחברות ל-private registry. במצב כזה, כאשר ניצור POD והוא יצטרך לבצע pull ל-image שנמצא ב-private registry. נוכל לשייך לו את פרטי ההתחברות באופן מוצפן באמצעות secret.

לפני שנכניס מידע ל-secret נצטרך לבצע לו **encode** באמצעות הפקודה הבאה:

echo -n “*<value>*” | base64

הפלט של הפקודה הזו יהיה encoding של הערך שאותו נרצה לאחסן ב-secret.

קיימים כל מיני סוגים של secrets כאשר הסוג הבסיסי ביותר הוא: opaque



Extra:

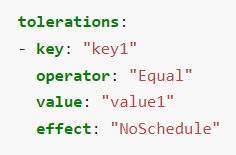
**taint & Tolerations**

taint זו אופציה שניתן להגדיר ל-NODE, לדוגמא:

kubectl taint nodes node1 key1=value1:NoSchedule

פקודה זו מציינת שאנו נותנים ל-node1 את האופציה: NoSchedule

שזה אומר ששום POD לא יוכל לרוץ על ה-NODE הזה אלא אם יש לו את ה-**toleration** הרלוונטי.



ואיך יראה ה-toleration הרלוונטי? ממש כך:

בקובץ הקונפיגורציה של ה-POD אנו נוסיף את המקטע הבא

שמציין שיש לו את ה-KEY וה-VALUE הרלוונטים, יחד עם האפקט של: NoSchedule.

וזוהי רק דוגמה אחת - קיימים מגוון סוגים של taints למגוון סוגים של מקרי שימוש.

**limits:**

ניתן להגדיר בקובץ קונפיגורציה של POD ואפילו של namespace, גבול כלשהו למשאבים שנרצה להקצות לו.

לדוגמא: נרצה ש-POD מסוים יתחיל עם מספר מינימלי של זיכרון, למשל 64MB

אך לא נרצה שהזיכרון יחרוג מ: 128MB

היתרון של limits הוא שבעזרתו אנו שולטים על המשאבים ב-cluster שלנו.

**jobs:**

ממש כמו בלינוקס, גם ב-K8S קיימת לנו האופציה להריץ cron jobs.

איך זה בא לידי ביטוי?

זה למעשה container, בעל תפקיד מסוים, שמתחיל על ידי טריגר שאנחנו נחליט, וברגע שהוא מסיים את תפקידו הוא נמחק. עד לפעם הבאה בה ה-JOB הזה יצטרך לרוץ.

מה ההבדל בין job לבין cron job?

job ירוץ כאשר אנחנו נחליט להריץ אותו, לעומת cron job אשר ירוץ לפי מרווח זמן מסוים, למשל פעם בשעה.

**DaemonSet:**

אובייקט זה מוודא שכלל ה-NODES שנציין אצלו (יכול להיות כולם יכול להיות חלקם) יריצו את אותו העתק של POD.

השימוש העיקרי שלו הוא בעיקר איסוף לוגים מה-NODES.

**lens:**

K8S יכול להיות קצת מסובך, במיוחד אם ה-cluster שלנו מורכב מהמון אובייקטים, בחברות גדולות הוא יכול להגיע גם לאלפי אובייקטים.

אז כדי שנוכל לראות את התמונה הגדולה בקלות - הגיע lens.

זוהי אפליקציה שמתחברת ל-cluster שלנו ומציגה אותו בפשטות באמצעות web UI שנוח לעין.

ניתן לראות את כלל האובייקטים: pods, namespaces, nodes, network, replicaSet ועוד…

מצורף לכאן קובץ ההתקנה ל-windows.