Imgae

עבור תוכנה שמריצים, יש לרוב צורך בעוד run-time environment (פענוח השפה של התוכנה) ו-dependencies (תוכנות נוספות שצריך עבור ההרצה). הבעיה היא איך ניתן לשנע את שלושת המרכיבים ולהתקין אותם בקלות. Image הוא קובץ tar מקובץ שמכיל את כל הנ"ל ונבנה ע"י סט הוראות (dockerfile).

Containers:

פתרון של docker להרצה של תהליכים בצורה מבודדת ללא גישה למידע הנמצא מחוץ לקופסת ה-container.

Kubernetes

פלטפורמה להרצה של container-ים, המספקת יכולת ניהול יעילה.

היסטוריה:

VMware יצרו בשנות ה-2000 את ה-VM המסחרי, ובזאת הפרידו את ה-OS מהברזלים. ב-2010 דוקר יצאה לשוק, ואיפשרה את השימוש ב-isolated process לשימוש ע"י כלל המתכנתים. גוגל לקחה את borg (מוצר לפריסה וכו'), והחלה לשלב החל מ-2012 את הקובקנטיס, כפלטפורמה שתנהל את הדוקר. ב-2017 יצאה גרסת קוברנטיס ראשונה של production, אך הפתרון לא מתכלל את הכל, לודג' UI, דאגה לשרתים ואבטחת מידע. במקום יצאו סביבות מסחריות (openshift לדוג') שנותנים את שאר המעטפת הנצרכת.

רכיבי cluster Kubernetes

## CONTROLS

הדרך של k8s לעבוד היא בעזרת controller-ים, רכיבים שכל הזמן מחפשים להגיע ל-desired state מסוים. 'כל רכיב שמחכה לי – צריך לעבור טיפול שלא יחכה לי'

## רכיבים ב-control pane (master):

* ETCD מסד נתונים קל משקל, מצוין בשמירת קונפיגורציות. אם הרכיב תקול – כלום לא יעבוד.
* APISERVER שכבת api שעוטפת את מסד הנתונים. אף רכיב של קוברנטיס לא מתקשר ישירות עם ה-ETCD, אלא רק דרך ה-API. למשתמש יש פקודת kubectl, או Oc ב-openshift כדי לתקשר עם ה-API ולתת הוראות ל-master. ההוראות נשמרות ב-ETCD. אם הרכיב תקול – כלום לא יעבוד.
* SCHEDULER – מחכה ל-event של שינוי ב-ETCD שמחכה עבורו, ומבצע את השינוי הנדרש. למשל – הקמה של pod. הרכיב משנה את הסטאטוס מ-pending ל-scheduling, עד שיעביר את המשימה ל-worker. ההקצאה ל-worker מתבצעת ב-ETCD, על ה-worker יש agent בשם kubelet שסורק את ה-ETCD ומחפש משימות עבור ה-worker. אם הרכיב תקול – הסטאטוס יישאר בpending או scheduling.
* control manager – הרכיב שמוודא שכל דבר הגיע למצב הרצוי, לדוגמא: Node Controller דואג שה-node-ים ירוצו, ומבצע תיקונים אם הם בעייתיים. Endpoint Controller מנהל את אובייקטי ה-endppoint, שמחברים את ה-service ל-pod. Service Account & Token Controller מנהל את ה- Service Account וה-token-ים שלהם.
* Cloud Controller Manager (אופציונלי) – רכיב שקיים במקרה בו ה-k8s מנוהל על ידי רכיבים אוטומטיים אחרים, כדוגמת AWS או openshift. במקום לפנות בכל פעם ע"י ה-API, ה-CCM יושב בתוך ה-k8s מנטר את השינויים ומנהל את הקשר עם הרכיבים מהנהלים את k8s.

## רכיבים ב-worker:

* Container engine/runtime – תוכנה המתקשרת עם ה-kernel, ויוצרת isolated process ומנהלת אותם עבור ה-8s. (קרנל – שכבה המקשרת בין התוכנה לחומרה. ממומשת ע"י ה-driver-ים).
* kubelet - סורק את ה-ETCD ומחפש משימות עבור ה-worker. ה-kubelet הוא עובד בתצורת controller pattern, כלומר כל עוד מתרחש אירוע מסויים, תגיע למצב רצוי (desire state). ה-kubelet מריץ container engine ושולט דרכו ב-container-ים שהוא מריץ.
* Kube-proxy – שולט על התקשורת של ה-proces-ים, פותח וסוגר פורטים (כלומר אחראי על ה-tableroute).
* Nginx – ingress controller, כלומר רכיב שמנתב את התקשורת לתוך ה-service-ים המתאימים. הרכיב יושב ב-namespace משלו, קורא את קבצי ה-ingress, ומקבל את התקשורת ומנתב אותה לתוך ה-service-ים המתאימים, ו-egress שמוציא תקשורת החוצה.

CNI/SDN – LAN לוגי בין הפודים. התקשורת צריכה לצאת ל-NODE ולצאת דרך ה-NIC שלו, אבל לוגית מימשו לנו LAN שיחבר בין כלל הפודים

## אובייקטים בקוברנטיס

* POD – תא. מכיל container-ים. הסיבה לשים container-ים ביחד היא ששניהם חולקים את אותו ip address ואכסון, ומכירים אחד את השני כך שהם יכולים לתקשר ע"י "localhost" ומכילים את אותם משתני סבי. משמעות לדוג', שהם לא יכולים להיות באותו פורט,. משתמשים ב-container-ים משותפים אם התכולות צמודות, כמו שרת וה-cache שלו  
  pod מתחיל כ-pending עד שלפחות אחד ה-container-ים שלו התחיל לעלות, ואז הוא עובר ל-running. (לאחר שה-init containers שלו מסיימים לפי הסדר, מתחילים ה-container- ים הרגילים) אם כל ה-container-ים שלו סיימו בהצלחה – הוא עובר ל-succeeded, ואם משהו נכשל – failed. אם מוחקים אותו הוא נכנס למצב terminating.  
  לפודים יש resource request – כמה משאבים שהפוד מבקש ברגע זה, ו-resource limits – הגבלה על כמות המשאבים. אם פוד ינסה לעבור את ה-limit הוא יפול (403).

פוד יכול לדוס את פקודת ההרצה של ה-image והמשתנים, או לדרוס רק את המשתנים של פקודת ההרצה, על ידי הזרקת מערכים ל-command ול-args.

ע"מ לדבג מומלץ לשנות את restartpolicy ל-never, כך ה-k8s לא ירים מחדש פוד שנפל

* Namespace – הפרדה לוגית בין סביבות שונות, משאפר פודים עם שמות זהים ב-namespace-ים שונים, מה שהופך אותו לאידיאלי עבור הפרדת סביבות prod וכד'. הפרדה בין namespace-ים שונים מאפשר שימוש באותם השמות. אין אפשרות להעתיק אובייקטים בין namespace-ים שונים, חייב למחוק וליצור מחדש. גם המשאבים (CPU זיכרון וכו') ייחודיים לכל namespace
* Replica set (או RS)– נועד לעטוף את הפודים ולאפשר הקמה של מספר פודים זהים. מכיל שני רכיבים: pod naming- דואג לשמות לפודים שבאחריותו, pod config – קובץ הקונפיגורציה של הפודים. בתוך ה-spec של ה-RS, יש template, שהוא בעצם המידע של ה-pod. אם צריך למחוק פוד – יימחק הצעיר ביותר.  
  הדרך שבה ה-RS מזהה את הפודים היא ע"י שדה בשם matchLabels שנמצא ב-selector, שם מוגדרים labels בתצורת and (כלומר הפודים חייבים להכיל את כלל ה-labels שמוגדרים ב-matchLabels). ה-RS למעשה מבצע תמיד kubectl get pos -l [[labels]].  
  במחיקת RS יש מעין finalizer בלתי נראה שמוחק את הפודים.
* Deployment – מקביל ל-statefulset, אבל מכיל מידע עבור פודים שהם stateless, כלומר כל פוד יכול לתת שירות באופן עצמאי. האובייקט מבצע אבסטרקציה המיועדת להנגשה למשתמשים, הוא עוטף את replicaset, ומכיל מידע זהה ל-replicaset עבוד הפודים. ה-deployment יוצר RS עם ID שנוצר מה-hash של הקונפיגורציה של ה-deployment. כשמשנים ב-deployment את המידע של ה-replicaset, נוצר replicaset חדש עם מספר רץ, אבל ה-replicaset לא נמחק, אלא רק נפטר מהפודים שלו. המטרה היא לאפשר תחקור קל ויעיל של ההבדל בין גרסות deployment, כשלפעמים השינוי של ה-deployment כברירת מחדל מספר ה-replicaset שנשמרים אחורה הוא 10, וניתן לשנות. כשמשנים גרסה ניתן לקבוע אסטרטגיה בה החלפת הפודים תתבצע בצורה מבוקרת.
* Service – מאפשר גישה לפוד. כתובות ה-IP של פודים הם דינמיים, ובכל פעם פוד עולה עם IP שונה. ה-service מאפשר service discovery, כלומר גילוי של הפודים ע"י שימוש ב-DNS (הוא מגדיר את השם ב-DNS). ל-service יש כתובת IP מופץ ב-DNS בכל ה-nodes, והוא מכיר את הפודים הזמינים כרגע ע"י ה-lable שלהם. הוא מעביר את התקשורת ל-IP של פוד.  
  הקמה: kubectl expose deployment [[deplymentName]]. ניתן לעשות port redirection ג"כ.  
  סוגי service:
  + clusterIP: סרוויס שמאפשר גישה רק פנימית בתוך ה-cluster
  + NodePort: סורוויס שמנגיש את הגישה גם למחוץ ל-cluster, ע"י פתיחת פורט מסוג high port על כלל ה-node-ים. זה שימושי עבור backing service, כלומר שירות שאינו HTTP (כי צריך לציין פורט מוזר). ב-cluster-ים גדולים ניתן להגדיר שה-port ייפתח רק ב-worker שבו יש את ה-POD.
  + LoadBalancer: תוספת על NodePort, מאפשר לחבר LoadBalancer לפודים, במקום להיות תלוי בכתובת IP יחידה של ה-POD. שירות שניתן ע"י ספקים שונים ומבצע loadbalancer בצורה אוטומטית. ה-loadbalancer עצמו הוא לא של k8s. הסיבה לצורך ב-loadBalancer ב-service הוא מימוש שלל health Chack שבודק אם ה-NODE זמין והאם ה-port פתוח עליו. השירות החיצוני מבצע port-forwarding ל-node-port של nginx.  
    הוא יכול לעבוד בצורת round-robin, או session affinity (שמשאירה את המשתמש מחובר לאותו הפוד עבור ביצועי stateless) או custom.
  + externalName: משמש ע"מ לחשוף שירות שאינו נמצא על ה-k8s, במקרה כזה כשמנסים לגשת ל-service יוחזר cname שמוביל לשירות אי-שם מחוץ ל-k8s.

Service mesh – מאפשר תקשורת ישירה בין ה-sservice-ים במקום לעבור דרך ה-k8s, מאפשר אבטחה גבוהה יותר של המידע ומהיות תקשורת service-ים משופרת

* Ingress(/route)– קובץ הנקרא ע"י ה-nginx, וממפה נתיב לשירות. הנתיב יכול להיות שם הדומיין או החלק שאחרי ה-'/'. תומך ב-HTTP או HTTPS. אם הנתיב לא קיים – הלקוח יופנה ל-defaultBackend. הנתיב יכול להכיל '\*' שמשמעותו כל תוכן, יכול להיות מדויק או תחילית של הכתובת. אם ישנם מספר ingress תואמים – העליון יקבל עדיפות.
* IngressClass – קובע איזה controller יטפל ה-ingress הנוכחי. ניתן להוסיף פרמטים (או להפנות ל-CM המכיל פרמטרים) ע"מ להשפיע על התנהגות ה-controller ביחס ל-ingress הנוכחי. קיים default בו משתמשים ה-ingress שלא מצוין בהם ingressclass.
* DeamonSet – אובייקט מקביל ל-deployment, ההבדל הוא שאצלו יש פוד אחד(או יותר) על כל NODE (או על כל ה-node-ים שיש להם label מסוים), באופן קבוע. המטרה היא לשמור יכולת ניטור על כלל ה-NODE-ים.
* statefulSet – מקביל ל-deployment, אבל מיועד ל-statefull application, כלומר אפליקציות שאכפת להם מהסביבה, בעיקר DB, שהם חייבים לדעת מה שם המכונה וה-IP וכד'. הפי'צרים שניתנים פה הם:
  + סדר הדלקה וכיבוי
  + Naming convention
  + IP ספציפים
  + חיבור הפודים ל-PVC
* PVC – מיפוי של volume ממקור אכסון חיצוני לתוך ה-POD. ה-POD משתמש בנתיב כתקייה מקומית, אך בפועל ישנו אכסון, המקושר ל-k8s ע"י PV, ומיפוי של ה-PV ע"י PVC, שאותו מקשרים בתוך ה-deployment. בפועל, כאשר יוצרים PVC, ה-k8s יוצר באכסון מקום (PV) ומקשר אליו PVC אוטומטית.
* CRD – custom resource definition – יצירת סוג אובייקטים חדש. משמש עבור operator-ים (ע"ע)
* Job – controller שמייצר pod-ים בהתאם לדרישה, ומוחק את הפוד לאחר סיום הפעולה או בעצירה יזומה. אם הפודים נמחקו – הוא ייצר מחדש. נשתמש ב-job כשצריך הרצה חד-פעמית שתסתיים ב-complition. Deployment, deamonset ו-statefulset הם דוגמאות ל-job. Cronjob – יוצר job בהתאם לתזמון. ניתן לקבוע אם כל הפודים צריכים להצליח כדי שה-job יושלם, ע"י הגדרת completionMode. אם שלב נכשל (pod נכנס ל-failure) backoffLimit יגדיר כמה ניסיונות נוספים יתבצעו. podFailurePolicy מגדיר מה נחשב כישלון עבור ה-job. ניתן להגביל את זמן החיים של כל ה-job (activeDeadlineSeconds) וזמן שאחריו ה-job ימחק (ttlSecondsAfterFinished).
* Static pod – פוד המוקם ישירות על node מסוים, באמצעות ה-kubelet. ה-node יראה את הפוד ב-api, אבל לא יהיה ניתן לבצע בו שינויים משם. שימושי עבור node שאינו חלק מ-cluster. לפוד אין גישה למשאבים אחרים (כמו secret וכד'). היצירה נעשית ע"י יצירה של yaml פוד על ה-node, ומחיקה ע"י מחיקת הקובץ. לאחר שינוי מרסטים את ה-kubectl.
* Quota – ע"ע namespace, pod. סוגי quota: CPU, זיכרון לפודים, אכסון לPVC, מספר אובייקטים מקסימאלי (אפשר לקבוע לפי כל סוג של אובייקט) סינון scope – מצב האובייקט, מאפשר גם מסננים מותאמים כמו lable של priority, או אכיפה של כללים (אם שמים על הגדרה quota של 0). ה-quotaלא משקף את כמות המשאבים האמיתית.
* LimitRange – מגביל את ה-quota שכל אובייקט יחיד יכול לבקש, מינימום ומקסימום, ומגדיר ברירת מחדל (מה שלפעמים בעייתי אם מגדירים רק request, אבל ברירת המחדל של limit נמוך יותר). לא מגביל פודים שכבר רצים.
* AdmissionConfiguration – אוכף הגדרות ב-k8s, משמש לרוב לאבטחה, מיטוב סטנדרט האובייקטים וטיפול בלוגים.

## סכמות אובייקטים:

* Version: הסכמה של האובייקט, שלפיו הk8s מפענח את האובייקט
* Kind: סוג האובייקט
* Metadata: מידע גנרי על האובייקט
  + Lablel: מידע המאוכסן בצורת key-value, ומתאר את האובייקט. לרכיבים אחרים יכול להיות selector, שמתייחס רק לאובייקטים שבהם מופיע/ות ה-label value המסוים
* Spec: הקונפיגורציה של האובייקט
  + משתנים ב-YAML

## פקודות:

* לכל האובייקטים יש get, create, update, delete. Describe כדי לקבל בצורה קריאה את כל הפרטים.
* ניתן להגדיר -o wide לכל פקודה, על מנת לקבל ב-output פרטים נוספים.
* ניתן להוסיף -w לפקודות השליפה עבור watch, שיעדכן אותי בכל פעם שמתבצע שינוי.
* אם מוסיפים -l – מקבלים את הפודים עם ה-lable המסויים הזה.
* Logs [[objectname]] – מביא את הלוגים על אובייקט מסויים
* Rollback- קיים בחלק מהאובייקטים (deployment), מבצע update לגרסה הקודמת שהייתה פעילה

## volumes:

Volume הוא שטח אכסון עבור ה-container-ים, עבור כלל השימושים. ה-volume תמיד יישב יחד עם הפוד על אותו ה-NODE. ניתן לשתף את ה-volumes בין ה-containet-ים, ואף בין ה-pod-ים, אבל זה לא מומלץ.

סוגים:

* hostPath – שטח אכסון שיושב על ה-node עצמו, ומוקצה לשימוש ה-container, לא מומלץ לשימוש (עדיף להשתמש ב-PV)
* PV – persistent volume, שטח המוקצה עבור ה-k8s, על node, שרת מרוחק או ענן. ה-PV ניתנים לשימוש ע"י PVC – בקשה לגישה למשאבים, מכיל מידע על אפשרויות הגישה. ברגע שנוצר ה-PVC ה-k8s יוצר עבורו PV. ניתן לגבות PV ע"י volume snapshot.
* ephemeral volumes: קטגוריה המכילה מידע זמני:
  + emptyDir – תקייה ריקה, אם הפוד נופל – המידע יימחק.
  + CSI Ephemeral Volumes – שטח אכסון המסופק ע"י ספק חיצוני כ-drive, נחשב למאובטח יותר.
  + DownwardAPI – metadata או משתני סביבה המסופקים כ-read-only
  + configMap – קובץ הגדרות שנשאב מתוך ה-k8s api, יכול להיפרס כקובץ או כמשתנה סביבה.
  + Secret – מקרה פרטי של configMap עבור קבצים חסויים (לודג' סיסמאות, PKI). להבדיל ב-CM שנשמר כקובץ, ה-secret נשמר ב-RAM, הוא שמור כ-Base64, ויש לו הגבלת גודל לכ-1500 bytes. חשוב לשים לב לא לשים אותו כ-enviroment variable, ולא לשרשר אותו בשום צורה גלויה.

## Kubernetes autoscaling:

* Horizontal Pod Autoscaler – במקרה של שימוש גבוה במשאבים ע"י פוד – מעלה את מספר הפודים (ניתן לקבוע ב-deployment)
* Vertical Pod Autoscaler - במקרה של שימוש גבוה במשאבים ע"י פוד – מעלה את המשאבים לפוד
* Cluster Autoscaler – במקרה של חוסר במשאבים עבור הפודים – מעלה node-ים נוספים

## RBAC

השיטה ברירת המחדל לניהול הרשאות עבור ה-k8s. הוא מכיל 4 סוגי אובייקטים:

* Role – אובייקט המגדיר הרשאות בתוך namespace ספציפי, מכיל את: האובייקט המבקש (לא המשתמש אלא המקור ממנו הגיעה הבקשה), האובייקט המבוקש ואילו פקודות יכול המבקש לבצע על האובייקט המבוקש. אם האובייקט מכיל אובייקטים נוספים (כמו pod/log) – ההרשאה **לא** תחלחל. '\*' עבור הכל, וניתן גם להגביל שם של resource.
* ClusterRole – דומה ל-role, אבל מגדיר עבור כלל ה-clusrer. ניתן ליצור aggregation, שיאחד יחד כמה clusterRole, עבור נוחות שימוש.
* roleBinding – מחבר role למשתמש, קבוצה או ל-service account.
* clusterRoleBinding - מחבר clusterRole למשתמש או ל-service account.

האובייקטים המשלימים הם משתמש, קבוצה ו-serviceAccount שמקבלים את ההרשאות. קיימות קבוצות ברירת מחדל לכל מיני סוגי אובייקטים.

# Nodes:

## taint and toleration

taint – הגדרה על node, שימנע מלהריץ פודים, אלא אם יש להם toleration זהה להגדרת ה-taint, ואז מותר להם לרוץ. ניתן להימנע מהרצת פודים חדשים, לחסום הרצת פודים חדשים או להפיל את הפודים הרצים.

Toleration – הגדרה ב-spec של הפודים, יכול להיות או זהה לחלוטין, או זהה ברמת ה-key.

## Affinity:

מאפשר קביעת ארכיטקטורה למען מיטוב ה-pod-ים. אם מתייחס לפודים ב-namespace אחר מכונה crossNamespace

Pod affinity – מאפשר קביעה עם אילו pod-ים נוספים חייב לשבת ה-pod שלי, למען שיפורי ביצועים.

Pod anti-affinity - מאפשר קביעה אילו pod-ים נוספים אסור להם לשבת ה-pod שלי, למען שיפורי שרידות.

Node affinity – קביעה על איזה סוג של node ישב ה-pod שלי, כאשר הקביעה של ה-node מתבצעת ע"י ה-lable של ה-node.

## שיפור שרידות:

על מנת לשפר את שהשרידות של ה-k8s, מתבצעות מספר פעולות:

* Load balancer- הפנייה ל-k8s מתבצעת באמצעות LB
* יצירת מספר control plane ו-worker-ים
* Fault tolerance – וידוא שהפודים תקינים ע"י replicaset, בדיקת תקינות של הפודים ע"י health check ו- Node failure handling - העברת ה-pod-ים ל-node תקין במקרה של תקלה ב-node

# משתמשים ואבטחה

## Human account

## Service account

משתמש המשמש לצורך הזדהות מול רכיבים ב-k8s, אך מיועד פנימית בתוך ה-k8s לרכיבי pod. כברירת מחדל, כל namespace יוצר משתמש serviceaccount בשם default, ובו משתמשים הפודים, אלא אם הוגדר בפוד אחרת (בוטל או משתמש אחר).

Serviceaccount הוא ייחודי בתוך ה-namespace, אבל ניתן להגדיר ב-role binding הרשאה עבור service account מ-namespace אחר.

Security context

מקטע שמוגדר ב-spec של פודים (או בתוך המקטע של container), הוא יכול לדרוס את המשתמש בו רץ ה-container למשתמש/קבוצה אחרת. השינוי לא משפיע עבור גישה ל-ephemeral volumes.

אפשרות נוספת היא להשתמש ב-capabilities, שמגדיר פקודות מותרות/אסורות לשימוש, אף שהמשתמש לא משתנה ל-root. בנוסף ניתן לשלוט על Seccomp ועל SELinux (יכולות אבטחה של לינוקס)

# כללי

## אבטחה:

ישנם מספר רכיבי אבטחה המסייעים להגן על תוכן ה-k8s:

* RBAC – role base access control – מי רשאי לגשת לאיזה משאב.
* Network policy – מנהל את התקשורת של הפודים, ומי רשאי לתקשר עם מי.
* Pod Security Policies- deprecated, הוחלף ב-RBAC
* Secrets – שמירת מידע מוגן תחת secret ע"מ למנוע גישה לקריאת המידע
* כמובן שתוכן הפודים צריך להיות מאובטח, עם שימוש בהרשאות מתאימות בתוך ה-container, כמו בכל מערכת הפעלה.

## Helm chrat

על מנת לארוז את כלל חלקי הפרויקט יחד, יצרו פרויקט שנקרא , שיודע לארוז ולדלוור את כל הפרויקט יחד. ה-helm זקוק ל-kubectl (או oc), כדי לרוץ, והוא שואב את הקונפיגורציה מתוך \.kube\config שזה הנתיב של הגדרות ה-k8s. יש באינטרנט מלאי ש helm שזמיני להתקנה במיידית.

פקודות ה-helm

* Helm create – יצירת מבנה התיקיות עבור פרויקט helm חדש (builer plate).
* Helm template . – בונה את התיקייה הנוכחית לקבצים בצורתם הסופית
* Helm package. - בונה את התיקייה הנוכחית לקובץ tgz, תיקייה מקובצת שמכיל את כלל המידע
* Helm install [[instace name]] [[file]] – מקבל tgz ובונה אותו ב-k8s עם השם שהוזרק לתוך ה-k8s.
* Helm upgrade [[instace name]] [[file]] – מקבל tgz ומשרדרג את ה-helm עם אותו עם השם שקיים ב-k8s. ניתן להשתמש ב-'-f' כדי להוסיף קובץ שידרוס את ה-value.yaml

קבצי ה-heml

* Values.yaml – הקובץ שמכיל את המידע עבור הבנייה
* Chart.yaml – מכיל את המידע על הפרויקט (מהו הפרוייקט וכו')
* Test – תקייה המכילה unitest, שירוצו מיד לאחר ה-helm install
* Template: מבנה ה-yaml-ים עצמם. קבצי ה-YAML מלאים ב-{{setting}}, ששואבים את מידע ממקור אחר. האידיאל הוא לא לשנות כלום במבנה הקבצים, מלבד ב-values.
  + {{.Values.something}} – שולף את הערכים עבור something מתוך ה-values.
  + {{if …..}} – תנאי if
  + {{with …..}} - loop על הערך המתקבל, עד לתג {{end}}
  + {{…|…}} – העברה של הלק שמאל לחלק ימין
  + {{ ….. |nindent X}} – את הערך המתקבל תזיח X רווחים ימינה
  + {{ ….. |default ….}} – קח את צד ימין אם שמאל לא קיים

Environment specific

הרציונל הוא הפרדה בין הגדרות אפליקציה/מערכת, כגון image,emptydir וכד', למול הגדרות סביבה, כמו גודל זיכרון, מס' פודים, כתובת DB.

## Custom value

הנגשה של עריכה של קובץ ה-value. מתבצע ע"י קובץ value חיצוני (לא קשור למבנה ה-helm), והצמדה לקובץ ה-tgz באמצעות הפקודה install/upgrade.

## Health-check

בין כל pod ל-service יש רכיב שנקרא probe, שמבצע בדיקות liveness/readiness ה-yaml של ה-probe הוא בעצם ה-yaml של ה-deployment. אם הבדיקה של ה-readiness נפלה – ה-service לא ינגיש את הפורט, אם הבדיקה של ה-liveness נפלה – ה-k8s יפיל את הפוד וינסה להרים אותו.

יששנם סוגי בדיקות שונות כמו HTTP שמחכה לסטאטוס 200, או tcpsocket שרק רוצה תקשורת TCP בפורט מסויים, ויש שמבצעים exec – פקודה בתוך ה-pod. הבדיקות יכולות להמתין זמן מסוים מההתחלה (initialDelay), וזמן לבדיקה מחזורית (periodSeconds).

לרוב liveness יבדוק פורט מרכזי, וייקח initialDelay ארוך יחסית, ע"מ לאפשר לתחקר אם משהו כל הזמן קורס. Readiness יבדוק מכלול של זמינות (thread-ים, זיכרון וכוד'), וינגיש את הבדיקות בפורט אחר, אותו לא אחשוף החוצה כלל.

## Pod Disruption Budgets

הגדרה הקובעת מהו המינימום של הפודים שחייבים להיות זמינים בכל עת, בזמן הפצות ועדכונים של הפודים.

Pod priority and preemption:

במקרה שחסרים משאבים ל-cluster, ה-k8s יבדוק לפי ה-priority אילו פודים להעלות ואף ישמיד (preemption) פודים זניחים.

## Operator

ה-k8s מאפשר ליצור סוגי ישויות חדשים, עם controller שיטפל בסוג הישות החדשה. בשביל ליצור דבר כזה יוצרים יישות מסוג custom resource, שמגדירה את הסכמה של הסוג ישות. ישנם SDK (סביבת פיתוח לפרוייקט מסוג ספציפי), כשהמוביל בהם הוא של GO. ה-SDK הופכים ל-image שמהווה controller עבור ה-CRD.

## ArgoCD

תצורת עבודה בה ה-source of truth הוא ה-git. ה-git מעדכן באמצעות pull (ידני או מחזורי) את ה-argocd, שמעדכן בצורה אוטומטית את ה-k8s, מקרה פרטי של אופרטור מוכר. ה-ArgoCD מגיע כ-helm (כברירת מחדל הסרויס מוגדר כ-clusterIP, אם רוצים לפתוח צריך לשנות ל-LoadBalancer), עם סיסמה ראשונית ששמורה כ-secret (כמובן ב-base64).

ע"מ לפרוס אפליקציה ב-k8s שאנחנו יושבים עליו, משתמשים ב-kubernetes.default.svc. כלומר ל-svc שנמצא ב-default ונקרא Kubernetes, שהוא השער של ה-k8s.

ניתן לציין קובץ value מסויים כדי להפריד בין סביבות: מייצרים קבצי value שונים עבור כל סביבה, וב—Argo משתמשים בקובץ ה-value המתאים לסביבה.

## Deprecated:

Replication Controller מוודא שמספר הפודים הרצוי זמין. חלק מ-control manager.

# מושגים

* Taints
* Tolerations
* Pod status:
  + Pod status - NoSchedule
  + Pod status - PreferNoSchedule
  + Pod status - NoExecute
* PV
* Hostpath
* Emptydir
* CSI Ephemeral Volumes
* DownwardAPI
* emptyDir
* ephemeral volumes
* volume snapeshot
* ingress
  + ingress – exect
  + ingress -prefix
* service
* pod
* node
* cluster
* namespace
* defaultBackend
* Nginx
* ingressClass
* init container
* pod definition:
  + pod-env
  + pod – envFrom
  + pod-command and argument
  + pod restart policy

מהו platform:

אנחנו ב-appdev, כלומר אנחנו רוצים לפתח אפליקציה, אבל אנחנו תלויים ב-3 דברים:

* Beacking services: - אפליקציות שאנחנו משתמשים בהם במוצר שלנו, לדוג' DB, S3 ועוד תוכנות שיושבות מאחורי הקלעים
* Tools – כלים שסביבת הפיתוח תלויה בהם, לדוג' source control management (git), automation (Jenkins), knowlagebase(confluence).
* Infra: הסביבה שעליו יושב המוצר, לדוג': מערכת הפעלה, k8s, סביבה וירטואלית..

שלושת אלה נקראים platform. כשמנסים ליצור מוצר – צריך לדאוג גם לכל זה. אם המוצר מגיע ישר ללקוח (כמו אתר אינטרנט) אפשר לרכוש אופציות זולות במקום לפתח ולטפח את כל הנ"ל. אך אם המוצר מגיע לארגונים, אני צריך לדאוג לכל הנ"ל כחלק מהמוצר עצמו. ככל שאנחנו מוצר יותר קטן – נשתדל לייצא את כל הנ"ל למקורות חינמיים פשוטים.